Modbus RTU for Embedded Cyber Secure Inverter Controller

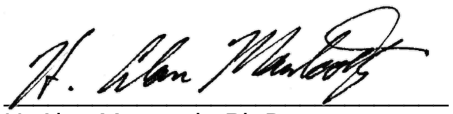
A thesis submitted in partial fulfillment
of the requirements for the degree of
Master of Science in Electrical Engineering

by

Brady McBride
University of Arkansas
Bachelor of Science in Engineering, 2021

May 2023
University of Arkansas

This thesis is approved for recommendation to the Graduate Council.



H. Alan Mantooth, Ph.D.
Thesis Director

Chris Farnell, Ph.D.
Committee Member

Jia Di, Ph.D.
Committee Member

Abstract

The Modbus communication protocol is a widely adopted communication standard in industrial control systems. This communication protocol is known for being reliable and straightforward to implement while being versatile in terms of its operating parameters while supporting multiple formats over various hardware infrastructures and architectures. Many intelligent devices such as Programmable Logic Controllers (PLCs), Human-Machine Interfaces (HMIs), Internet-of-Things (IoT), and various Operational Technologies (OT) utilize Modbus for their communication systems. These types of systems must communicate with each other through a standardized and central communication process. To support the integration of these modular systems, a Field-Programmable Gate Array (FPGA) can act as an embedded central routing fabric for this communication to take place.

Embedded systems are versatile enough to interface with various devices and systems to accomplish various goals. Additionally, embedded systems require relatively small physical designs to minimize the required resources to facilitate the intended application by providing low-level system access. This minimization of system resources goes hand in hand with reducing the financial cost of a proposed solution or system. As remotely collaborating researchers often use FPGAs to prototype designs that are required to have a method for data transmission among systems, it is imperative to provide a baseline standard for communications among devices and systems.

A typical method of implementing the Modbus RTU communication protocol in an embedded environment is using integrated logic architectures within the FPGA called “Intellectual Property (IP) cores.” IP cores can be designed using integrated logic or circuit designs to function as an embedded processor. These IP cores can then perform the required computational actions to support the Modbus RTU communication protocol by utilizing high-level programming languages such as the C programming language.

The hardware description language of Very High-Speed Integrated Circuit Hardware Description Language (VHDL) allows for the control of real hardware at the logic gate and signal level. These logic gates and signals can be designed and controlled to perform desired actions based on the system

design. Programming an FPGA using VHDL allows an individual to access the lowest abstraction level of the system during FPGA development. This level of abstraction is referred to as the register-transfer level (RTL), which gives access to manipulating values and variables at the register level. This register-level manipulation provides precision over creating the logical circuit within the FPGA, thus minimizing the required code to perform desired operations.

The Modbus RTU communication protocol can be implemented within an FPGA using VHDL programming to establish a standardized and embedded serial communication pathway. This implementation provides a standardized communication protocol to streamline research efforts among researchers, thus increasing the efficiency of research efforts. Additionally, this Modbus RTU implementation requires fewer resources when compared to typical communication protocol implementations that utilize an IP core, reducing the hardware requirement for effective research efforts.

Acknowledgement

I am filled with immense gratitude and appreciation towards Dr. H. Alan Mantooth, my advisor and committee chair, for giving me the incredible opportunity to work alongside him and his research group at the MSCAD Lab. I cannot thank him enough for the doors that he has opened for me and the invaluable guidance he has provided throughout my journey. Through his mentorship, I have gained a depth of hands-on experience that has exceeded my expectations and allowed me to stand on the shoulders of giants. I will forever be grateful for his generosity and support.

I am overwhelmed with gratitude towards Dr. Chris Farnell for his unwavering support and guidance during my time as a member of the Cyber-Physical Security section of the MSCAD Lab research group. Under his mentorship, I was given the freedom to develop my skills and capabilities independently, while also benefiting from his vast experience and valuable perspective. Dr. Farnell's influence has been pivotal in shaping my problem-solving abilities and has helped me grow both personally and professionally. I cannot thank him enough for the significant role he has played in my development and success.

I am deeply grateful for the immense support and opportunities that Dr. Jia Di has provided me. His unwavering encouragement and belief in my abilities have been a constant source of motivation and comfort, inspiring me to pursue a career path that I am truly passionate about. I cannot thank him enough for the invaluable opportunity he has given me by admitting me into the CyberCorps: SFS program. His generosity and support have been instrumental in shaping my future, and I will always cherish his mentorship and guidance.

I am immensely grateful for the financial and professional support provided by the National Science Foundation's CyberCorps: Scholarship for Service award, which made this endeavor possible. I feel deeply honored to be a part of this distinguished program, which has given me the opportunity to work on some of the most challenging issues related to our nation's critical infrastructure. I will always cherish this experience and remain grateful for the support that enabled me to participate in this program.

I would like to express my heartfelt gratitude to Paulo Custodio for his invaluable support and guidance during this project. Paulo has been an exceptional team member, consistently providing constructive

feedback and insights that have greatly contributed to the success of my research efforts. Working alongside him has been a valuable learning experience, and I am thankful for the knowledge and expertise he has shared with me.

Finally, I cannot express enough gratefulness to my supportive family, especially my loving spouse, who stood by me unwaveringly throughout the years of completing this work. Their unconditional love and understanding gave me the strength and motivation to keep going, even during the toughest moments. I am deeply thankful for their unwavering support, which played a significant role in my success. I look forward to embarking on new adventures with my wife, as we close this chapter and move forward together.

Table of Contents

List of Figures

List of Tables

List of Abbreviations

ADC	Analog to Digital Converter
ASM	Algorithmic State Machine
CBA	Common Bus Architecture
CCS	Code Composer Studio
CLB	Configurable Logic Block
CRC	Cyclical Redundancy Check
CPLD	Complex Programmable Logic Device
DER	Distributed Energy Resource
DOE	Department of Energy
DT	Digital Twin
EBR	Embedded Block RAM
EEPROM	Electrically Erasable Programmable Read-Only Memory
FIFO	First-In-First-Out
FPGA	Field Programmable Gate Array
HDL	Hardware Description Language
HMI	Human Machine Interface
ICS	Industrial Control System
IDC	Insulation-Displacement Contact
IP	Intellectual Property
I/O	Input/Output
LUT	Lookup Table
MBAP	Modbus Application Protocol
MUX	Multiplexer
NCREPT	National Center for Reliable Electric Power Transmission
NSL	Next State Logic
PCB	Printed Circuit Board
P&R	Place-and-Route
RAM	Random Access Memory
RPi	Raspberry Pi
RTU	Remote Terminal Unit
SCADA	Supervisory Control and Data Acquisition
SETO	Solar Energy Technologies Office

SoC	System-on-Chip
SPI	Serial Peripheral Interface
TCP	Transport Control Protocol
TI	Texas Instruments
UCB	Unified Controller Board
VHDL	Very High-Speed Integrated Circuit Hardware Description Language

Chapter 1

Introduction

1.1 – Motivation

The modern power grid is an essential component of our critical infrastructure that plays a significant role in contemporary society. Our reliance on the availability and reliability of the electric utility grid makes it an essential component to facilitate the needs of individuals, such as performing a role in providing access to food, clean water, and sanitary conditions. Beyond servicing critical needs, the power grid is in virtually every amenity and aspect of our daily lives. As society continues to grow and develop, the demands on the electric power grid continue to increase. Additionally, with more substantial considerations of where this power is sourced, the transition to alternative sources, such as renewable energy, is continually in demand [1]. This newfound transition to renewable energy is often achieved with distributed energy resources (DERs), which naturally require network communication and coordination with the larger power grid. This, combined with a push for a “smart grid” to service society’s electrical needs with a more efficient and resilient electric grid, has increased the complexity of this critical infrastructure. To address the dynamically changing demands of the modern power grid, there has been a rise in the addition of network-connected grid devices [2].

While the addition of communication and networking infrastructure into the electric grid supports a more effective and efficient system, this change also introduces additional vulnerabilities to which these systems were not previously prone. Cyber attackers can utilize these networks to traverse once-isolated systems and gain access to previously “untouchable” critical infrastructure. As more critical components of the electric grid are connected to various networks, a resulting increase in the occurrences and the potential threat for cyber-physical attacks is undesirable [3]. In response to the looming threat of cyber-attacks on society’s critical infrastructures, governments such as the federal government of the United States have increased funding towards critical infrastructure improvements to enhance the electric grid’s resiliency and cyber defense capabilities [4]. To effectively address both the cyber and physical demands at a dynamic level, researchers need focused and effective efforts to respond to these problems.

As the growing availability of improving technologies is coupled with the widespread efforts of retrofitting the electric power grid to meet the ever-increasing energy demands, improvements in how these evolving technologies and techniques are utilized must be introduced. As the market transition to alternative energy sources arises, so does the necessity for cost-effective and impactful power and cyber-physical research solutions and methods to this problem. Money and time are two of the most imperative and finite resources required to address these issues. To continue creating more reliable and resilient critical infrastructure, adequate financing to support these efforts is essential. The time required to foster these innovations is arguably equal to the funding of these research efforts is the time required to foster these innovations. As the threat of cyber-attacks is consistent and growing, it is imperative to respond proactively and with swiftness.

To address the challenges of substantial research investments regarding money and time, improving efforts to lower solution costs while increasing collaborative research productivity is paramount. As technological advancements in the electric power grid require improved or additional electronic hardware, one method of reducing financial expenses is to utilize equipment with a lower fiscal cost. The drawback often seen with the reduction in the price of more affordable equipment and hardware is a comparable reduction in these devices' capability and computational resources. Thus, a lower-cost implementation requires research solutions that function on hardware with lower computational capabilities. Presenting research solutions that minimize the required computational and hardware resources effectively addresses the constant challenge of limited fiscal resources. The second finite resource is time. Bad actors are always persistent in their efforts to improve their cyber-attack capabilities. To be proactive in meeting these often-restless adversaries, vital considerations to the development time for research solutions must be made. By standardizing components of the research efforts, an increase in progress efficiency among collaborators is possible. In research, multiple entities often collaborate on the same project while being geographically distanced. By providing standardized solutions and components, researchers have a common platform during the design and development phase that promotes rapid prototyping between collaborators.

The Modbus RTU communication protocol is widely adopted within Industrial Control Systems (ICS). This communication protocol is commonly used as a standardized method of communication as it is open-source and known for being reliable. Embedded systems, such as FPGAs, are often used in ICS applications [5]. These embedded systems can integrate serial communication capabilities, such as Modbus RTU. An embedded processor architecture is a typical method of providing embedded systems with Modbus RTU functionality. These embedded processor architectures are often called “Intellectual Property cores” or “IP cores.” Embedded processors utilize logical algorithms to describe the embedded processor architecture and essentially function as a processor inside an embedded system, such as an FPGA. These embedded processors are then used to execute code from higher-level programming languages such as C. Such an implementation of the Modbus communication protocol using the ARM Cortex-M0 embedded processor can be seen in [6]. A similar method of Modbus RTU implementation within an embedded Linux system can be seen in [7]. These researchers continued their work to develop a data acquisition and monitoring system can be seen in [8] that is based on the embedded processor-operated Modbus RTU of [7]. The work presented in [9] is an example of using an embedded processor to utilize the Modbus RTU communicational protocol to control an industrial plant (crop) factory system. The work in [10] utilizes an ARM Cortex A-9 embedded processor to utilize Modbus RTU to manage the communication in their production control system.

This thesis presents the design and development of a Modbus RTU communication protocol implementation using VHDL within an FPGA that does not require an embedded processor. Thus, the required resources to utilize the Modbus RTU communication protocol are less when compared with the typical industry method of using an embedded processor. This reduction in required resources addresses the concern of cost-effective and resource-efficient solutions for communication within embedded systems. This custom Modbus RTU communication protocol resides within a low-cost FPGA mounted within a Unified Controller Board (UCB) to provide standardized communication to support multiple modules within the controller architecture for a multi-level cyber-hardened solar inverter. This approach provides a standardized serial communication protocol that operates as the routing fabric and communication between cyber-defense-related modules within the controller of a solar inverter. Implementing the Modbus RTU communication protocol was successfully developed while keeping the

required space smaller than typical embedded processor-based implementations. This work will discuss the design and development of an embedded Modbus RTU implementation designed to work within a low-cost FPGA while providing researchers with a reliable and resilient communication protocol to support data transmission.

1.2 – Solar Energy Technologies Office Project Contribution

The work presented in this thesis contributes to a greater project sponsored by the Department of Energy (DEO) Solar Energy Technologies Office (SETO). This SETO project is named “Multilevel Cybersecurity for Photovoltaic Systems” and addresses cybersecurity at both the inverter and system levels for photovoltaic energy systems. The embedded Modbus RTU implementation in this thesis contributed to this SETO project by providing a common and standardized data transmission point throughout the inverter controller architecture. The Modbus RTU implementation controls the FPGA’s routing fabric, giving access to shared memory for multiple modular cybersecurity-focused components within the inverter controller architecture. Further details regarding the SETO project and its relevance to this thesis’ contents will be discussed in Chapter 3.

1.3 – Thesis Organization

The list below sequentially outlines the contents and structure of this thesis:

- Chapter 1 discusses the motivation for designing and developing a Modbus RTU implementation using VHDL within an FPGA to provide a standardized serial communication protocol that utilizes fewer hardware resources.
- Chapter 2 defines the background concepts and fundamental knowledge considered in the design process. Additionally, this chapter discusses additional information that may be pertinent to the reader, such as topics related to hardware, communication protocols, and coding.
- Chapter 3 explains the system components and the design and development for creating the custom Modbus RTU implementation. Additionally, this chapter will discuss in further detail the involvement and application of this Modbus RTU implementation concerning the greater efforts of the SETO project.

- Chapter 4 demonstrates the experimental test setup and results of the various testing phases for this work. This chapter will discuss testing involving simulations, hardware, application, and the resiliency of this implementation.
- Chapter 5 concludes this design and development of the Modbus RTU communication protocol within an FPGA. This chapter also discusses additional steps taken to increase the throughput of this communication implementation. It outlines recommendations and actions to use encryption and message authentication techniques to secure packet contents when transmitting data.

Chapter 2

Background

2.1 – Introduction

As mentioned in the previous chapter, the primary motivation for this work is to provide a method of standardized communication to assist with improving the effectiveness of research efforts while simultaneously reducing the associated costs. As the risk of cyber-physical attacks and events is a constant and ever-moving target, it is imperative to be proactive and adequate to address these threats. The solutions presented by researchers and industry professionals must be current and conscious of these changing challenges and the limited resources available to manage them. These contemporary concerns are addressed regarding communication systems by implementing the Modbus RTU communication protocol within a standardized embedded device that requires fewer resources than typical embedded methods. The remainder of this chapter will discuss the concepts of the Modbus RTU communication protocol and the pertinent information regarding serial communication, associated hardware, and power electronics.

Additionally, the work presented in this thesis is in conjunction with a greater project focused on improving the cyber-physical security of power grid congrid-connected photovoltaic energy generation farms. This project is funded by the U.S. Department of Energy (DOE) Solar Energy Technologies Office (SETO) and is titled “Multilevel Cybersecurity for Photovoltaic Systems” (Award #DE-EE0009026).

2.2 – Power Electronics and SMA Inverter

As mentioned in the introduction of this section, the work presented in this thesis covers the contributions of developing the communication protocol implementation for developing a cyber-hardened photovoltaic inverter. The solar inverter used in this research project is an SMA Sunny Highpower PEAK 3 utility grid-connected photovoltaic (PV) inverter (henceforth referred to as an “SMA inverter”). This SMA inverter is a 60 Hz commercial off-the-shelf grid-connected PV inverter rated for 1500 volts of direct current (VDC) for 125 kW of power [11].

The SMA inverter is integrated with a Modbus interface that allows the device to communicate with various modules utilizing the Modbus communication protocol. This SMA inverter supports the Modbus TCP and Modbus UDP configurations of the Modbus communication protocol. Modbus TCP allows for both the reading and writing of registers, while Modbus UDP only supports the writing of registers. While the work presented in this thesis discusses developing a Modbus implementation using the Modbus RTU configuration, the existing hardware in the controller architecture allows for the conversion of Modbus RTU to that of Modbus TCP. This topic will be discussed in more detail in Chapter 2.5 and Chapter 3.2.1.

The SMA interface supporting the Modbus communication protocol was a significant determining factor in developing a Modbus RTU communication protocol implementation for this project. Modbus is a standardized communication protocol widely used and supported in applications such as distributed industrial control systems and inverter-based technologies [12].

2.3 – Serial Communication

Serial communication is a method that sequentially transmits data over a communication bus, bit-by-bit, as seen in Figure 2.1. This data transmission configuration only requires a few wires to send signals between devices. Thus, the infrastructure needed for serial communication is relatively low [13]. Within the family of serial communication, several communication standards exist. Two of the most common communication standards are RS-232 and RS-485. This project implements Modbus RTU under the RS-232 communication standard and will be the primary standard of discussion.

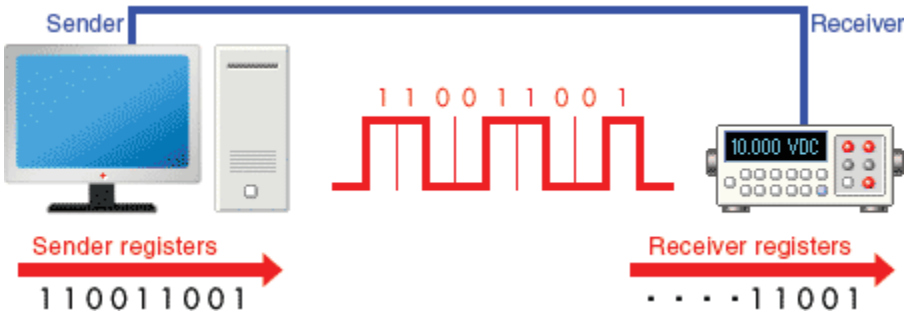


Figure 2.1. Sequential data transmission of serial communication [13]

Serial communication supports asynchronous and synchronous data transfer. In synchronous data transfer, the transmission of each bit of data is controlled by an external clock signal shared between the sending and receiving devices. This synchronization with a clock signal ensures that the data transfer across the communication pathway is coordinated and the interpretation of the data is read. This configuration requires additional infrastructure as the communication method's integrity relies on the synchronization between the sending and receiving devices [14]. As opposed to synchronous data transfer, asynchronous data transfer requires less infrastructure to implement as it is not reliant on a clock signal to orchestrate the transfer of data. Asynchronous data transfer sends a serial bit stream at a predetermined transmission rate. The message is then interpreted by the receiving device while reading the data bit-by-bit at the predetermined transmission rate. This data transmission rate is called the "baud rate" and represents the number of bits transmitted per second [14].

2.3.1 – RS-232 Serial Communication Standard

The RS-232 communication standard outlines the signals connected from the transmitting and receiving devices and characterizes the signals' electrical characteristics. The serial data exchange in the RS-232 standard protocol is carried out by controlling the voltage level on the communication bus and the rate the data is transmitted. To indicate a binary '0' or binary '1' signal on the communication line, the voltage level is modified with respect to a ground reference. The binary value of '0' is asserted when the voltage level of +3 V to +13 Vdc is read on the line, and a binary value of '1' is declared when the voltage level is -3 V to -13 Vdc [14]. A diagram of the voltage-binary logic can be seen in Figure 2.2

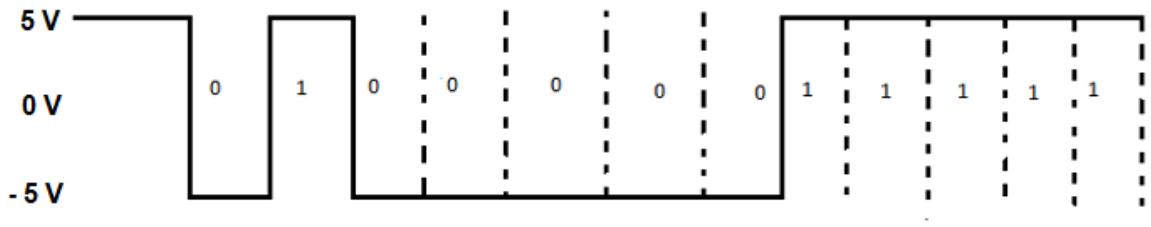


Figure 2.2. Voltage-Binary logic of RS-232 communication protocol [14]

The most basic wiring configuration to carry out serial communication requires three signal connections: Transmit (TX), Receive (RX), and Ground (GND). This wiring configuration can be seen in Figure 2.3. The wire connection from the transmitting device's TX port is fed into the port of the receiving device's RX port and vice versa. The voltage level across these signal lines is then changed to indicate the previously mentioned binary '0' or '1' value depending on the voltage with reference to the shared ground wire. This communication bus infrastructure allows for a dedicated signal pathway for both directions from each device's perspective. This configuration is called "full duplex" and allows for the simultaneous sending and receiving of data on each bus communication line [13].

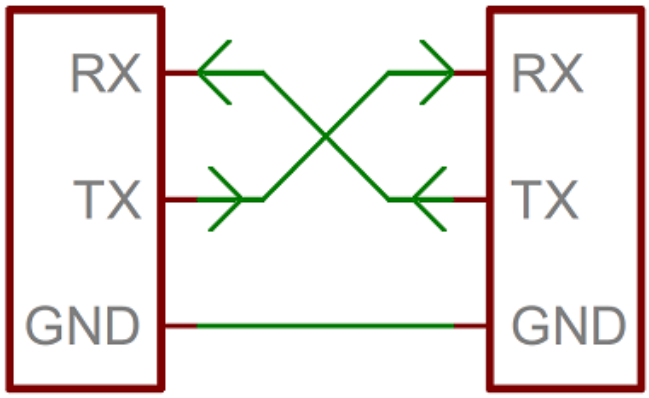


Figure 2.3. Three-wire serial communication connection diagram [15]

2.3.2 – Universal Asynchronous Receiver-Transmitter (UART)

Universal Asynchronous Receiver-Transmitter (UART) is used by embedded systems as a method for device-to-device hardware communication [16]. This system utilizes a three-wire setup to carry out the communication between devices. As this is an asynchronous system, a clock signal is not used to synchronize the output of transmission bits.

UART is configured to send data across a serial bus from one UART interface to another. A UART interface processes data in parallel, as seen in Figure 2.4, and sends that data across a serial bus line at a predetermined baud rate. As the data is sent sequentially, the UART system must frame the data in a manner that allows the system to determine when the start and stop of transmission are received. This process is referred to as "data framing" and consists of four components: start bit, data frame, parity,

and stop bit(s) [16]. These components together create the packet structure of the data transmission, as shown in Figure 2.5. A UART transmission line maintains a high voltage when waiting to receive data transmission. Then, it reads the incoming data when the line is pulled low by the incoming binary ‘0’ of the start bit. The size of the data frame is predetermined in the transmission configuration. Once the start bit is active, the UART device will process the incoming data at the predetermined baud rate. The parity bit allows the UART system to determine if the received message was error-free. To determine if the parity bit is to be a binary ‘0’ or ‘1’, the number of bits with a binary ‘1’ value is counted. If the number of binary ‘1’s’ in the data frame is even, then the data frame is considered to have “even parity,” and the even parity bit is asserted by the transmitting device. This same parity calculation is done by the receiving device to confirm that the calculated parity value during the transmission was correct. Finally, the stop bit returns the voltage level of the UART device to a high signal for one or two clock cycles (a predetermined number of clock cycles is used).

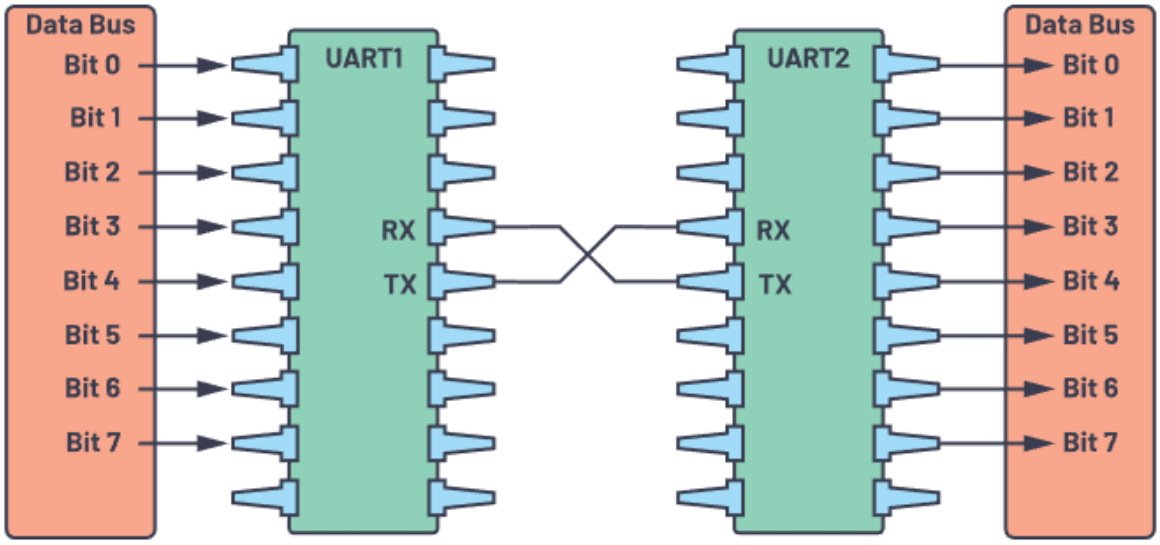


Figure 2.4. UART interface connection with Data Bus [16]

Figure 2.5. Packet structure of UART message [16]

2.4 – Modbus RTU

The Modbus RTU serial communication protocol is an open-source communication protocol widely adopted within the Industrial Control Systems (ICS) and automation industries. The packet structure of this communication protocol is relatively simple and is known for its reliability in carrying out serial communication. Because of the known reliability of this communication protocol, it is virtually supported in all communication infrastructure in the marketplace in the field of control systems and automation. The Modbus RTU communication protocol is typically implemented within an RS-232 or RS-485 interface [17].

The Modbus RTU communication protocol connects a supervisory or controlling device to a Remote Terminal Unit (RTU) within ICS and automation systems. This configuration is typical in supervisory control and data acquisition (SCADA) systems, where a master (client) device is responsible for communicating with one or more slave (server) devices to carry out data-based algorithms or controls. The high-level operation of the master and slave relationship consists of the master device sending a command to the slave device. This command is then deciphered, verified for integrity, and carried out by the slave device. These actions consist of two fundamental operations: reading and writing data. This data transmission may then be used to carry out control algorithms or functions of the system as intended.

The essential function of the Modbus RTU communication protocol is to connect various control instrumentation to logic controllers or to provide a data transmission pathway between two systems that are reliant on sharing data. The bulk of this thesis will focus on the latter, where the implementation design and goal are to be capable of transmitting data between various systems. Additionally, the scope of this work only requires the ability to read and write data to and from holding registers. Thus, the remainder of this background section will focus on the information pertinent to data movement between holding registers using Modbus RTU and any associated methods or communication protocol transformations.

2.4.1 – Data Representations and Configurational Parameters

Data transmission within the Modbus RTU communication protocol occurs at a predetermined baud rate and must be maintained at both the transmitting and receiving devices. Baud rates are typically standardized and occur at 9600, 19200, or 115200. This work was initially created to operate at a baud rate of 9600 bits-per-second. The potential to drive at higher baud rates, customize, and scale will be discussed in Chapter 5.

The Modbus communication protocol supports the transmission of two types of data: coils and holding registers. Coils represent a single bit of data, typically as a Boolean value of binary ‘1’ or ‘0’, respectively ‘ON’ or ‘OFF’ in an active high scenario. These coils represent the discrete status value of a system where an application may be a variable being asserted, indicating the status of a system. Holding registers represent 16-bit unsigned values capable of representing the hexadecimal values from 0x000 to 0xFFFF (0 to 65535) [17].

The first two bytes of a Modbus RTU packet contain the value of the device ID or the identifying value for which device the transmission is intended. If a device receives a message with a device ID matching its own, it carries out the command request from the master device. The value of the device ID can vary from 0x01 to 0XFF (1 to 255), where 0x00 (0) is reserved for a specific broadcast function. The broadcast function is intended for the master to send a command to all slave devices to be fulfilled by all devices.

2.4.2 – Read and Write Commands

The Modbus communication protocol uses a master and slave relationship architecture to perform the function commands based on the received function code. In this architecture, a master device is responsible for issuing orders to the slave device. The slave device responds to those commands and fulfills the requested operation. Modbus RTU supports eight different function operations that can be used to read or write from discrete coils or holding registers, denoted by their unique respective function codes. The eight supported function codes for these data types can be seen in Table 1.

Table 1 – Modbus RTU Function Commands

Function Code	Function Operation	Data Value Type
0x01 (01)	Read Coil	Discrete
0x02 (02)	Read Input Status	Discrete
0x03 (03)	Read Multiple Holding Registers	16-bit
0x04 (04)	Read Input Registers	16-bit
0x05 (05)	Write Single Coil	Discrete
0x06 (06)	Write Single Holding Register	16-bit
0x0F (15)	Write Multiple Coils	Discrete
0x10 (16)	Write Multiple Registers	16-bit

The remainder of this thesis will discuss the reading and writing of holding registers using the Modbus RTU communication protocol. This is due to the scope of work for the greater project only requiring access to data stored in holding registers. Additionally, as this thesis covers a custom implementation of the Modbus RTU communication in an embedded system, the required resources were able to be minimized by only containing a partial implementation of the Modbus RTU protocol. The utilized read and write function codes for this project are 0x03 and 0x10 which represent the actions to read and write multiple holding registers, respectively.

2.4.3 – Packet Structure

The Modbus RTU packet structure consists of two main configurations, read packets and write packets, while each packet structure can further be broken down into multiple components. A Modbus RTU packet's components are device address (ID), function code, address of a register in memory, number of registers requested, number of bytes more in the packet transmission, the data values, and the checksum.

The device address (ID) denotes the unique identifier for the slave device. Each slave device is programmed to only respond to requests explicitly made to itself. The function code is the operation carried out, such as reading or writing to holding registers. The address of the register in memory is the

value that indicates where the data being read or written is coming from or being written in memory. This data is separated into high (“Hi”) and low (“Lo”) bytes to form 16-bit data. The number of registers requested specifies how many registers are to be read or written to. The value for the number of bytes more indicates how many additional bytes of data will be following this value in the serial data stream. This is used to point to the device how much more data it is to expect to be coming through the serial transmission. The data values represent the data being read from memory or written to memory. This data is separated into high (“Hi”) and low (“Lo”) bytes to form 16-bit data. Finally, the checksum cyclical redundancy check (CRC) is a calculation performed by the master and slave device to verify message integrity.

In addition to the components of the reading and writing of holding registers, there are packet components for error handling. An error message is sent in response to a command request from the slave device back to the master device when a command cannot be carried out for any reason. The additional components of the error message are the modified function code indicating to the master that the slave device experienced an error and the error code indicating what the error experienced was.

Breakdown of Packet Structure

An example of the packet structure of a Modbus RTU transmission issuing a read command can be seen below in Figure 2.6. To break down each component of this packet structure, the remainder of this subsection will go over each component in sequentially highlighted figures.

Packet Component	Device Address	Functional Code	Address of First Register - Hi Byte	Address of First Register - Lo Byte	Number of Registers - Hi Byte	Number of Registers - Lo Byte	Checksum CRC - Lo Byte	Checksum CRC - Hi Byte
Byte (Hex)	11	03	00	6B	00	03	76	87

Figure 2.6 – Modbus RTU read holding register command

The first component of the Modbus RTU packet structure is the device address, which can be seen below in Figure 2.7, highlighted in brown. The device address component is an 8-bit unique identifier that is used to dictate which slave device a command is intended for. Each slave device has its

own unique device address and is programmed to only respond to messages addressed to its unique device identifier. The next component is the function code, or operational code, which can be seen below in Figure 2.7, highlighted in blue. This function code indicates to the slave device what the requested operation is. In this example, the requested operation is hexadecimal 0x03, which indicates the operation requested is a read command.

Packet Component	Device Address	Functional Code	Address of First Register - Hi Byte	Address of First Register - Lo Byte	Number of Registers - Hi Byte	Number of Registers - Lo Byte	Checksum CRC - Lo Byte	Checksum CRC - Hi Byte
Byte (Hex)	11	03	00	6B	00	03	76	87

Figure 2.7 – Read holding register breakdown 1 of 3

The next component of the packet structure is the address of the first register in the request, which can be seen below in Figure 2.8, highlighted in brown. This is a 16-bit value that indicates what the first address in memory is to index the read request. This packet component is sent in two 8-bit values representing the Hi and Lo bytes of the 16-bit value. The next component in the packet structure is the number of registers to read, which is shown below in Figure 2.8, highlighted in blue. This is a 16-bit value that indicates how many registers are to be read back to the master device. This packet component is sent in two 8-bit values representing the Hi and Lo bytes of the 16-bit value.

Packet Component	Device Address	Functional Code	Address of First Register - Hi Byte	Address of First Register - Lo Byte	Number of Registers - Hi Byte	Number of Registers - Lo Byte	Checksum CRC - Lo Byte	Checksum CRC - Hi Byte
Byte (Hex)	11	03	00	6B	00	03	76	87

Figure 2.8 – Read holding register breakdown 2 of 3

The final component of the Modbus RTU packet structure is that of the cyclical redundancy check (CRC). The CRC is a 16-bit value that is calculated and sent with the message contents. The calculation method of the CRC will be discussed in section 2.4.4. The role of the CRC is to confirm message integrity

and validity through a cyclical and robust algorithm of calculations and data modification. This packet component is sent in two 8-bit values representing the Hi and Lo bytes of the 16-bit calculated value. This packet component has its Hi and Lo bytes switched, where the Lo byte is sent before the Hi byte.

Packet Component	Device Address	Functional Code	Address of First Register - Hi Byte	Address of First Register - Lo Byte	Number of Registers - Hi Byte	Number of Registers - Lo Byte	Checksum CRC - Lo Byte	Checksum CRC - Hi Byte
Byte (Hex)	11	03	00	6B	00	03	76	87

Figure 2.9 – Read holding register breakdown 3 of 3

Read and Write Command and Response Structure

As the packet structure of a Modbus RTU request command is arranged in a certain manner, the response packet sent back from the slave device to the master device is similarly structured. While the read and write packets have similar packet components, there are some significant differences in the resulting message packet. The remainder of this subsection will discuss the request message and the responding message of both read and write commands, respectively.

First, the read command from the master device and the read response from the slave device for the Modbus RTU communication protocol are discussed. The packet structure of both the command and response can be seen below in Figure 2.10. From the figure, the read command sent from the master device is intended for the device with the unique device address of 0x11, and the function code of 0x03 indicates a request to read multiple holding registers. The address to index of the first register to be read is 0x006B, and the master device is requesting 0x0003 (3) 16-bit registers. Finally, the CRC is calculated and attached to the end of the Modbus RTU packet.

After this read holding register command request is sent from the master device, the slave device fulfills the request by providing data from the requested holding registers. The first two 8-bit components of the response packet contain the device address and function code repeated back from the original read request. The next component is an 8-bit value that represents how many more bytes of data will exist in the remainder of the packet transmission, excluding the CRC component. This portion is so the

master device knows how much data it is expecting to receive in the transmission. The CRC value is the final component attached to the packet structure of the read response.

Packet Component	Device Address	Functional Code	Address of First Register - Hi Byte	Address of First Register - Lo Byte	Number of Registers - Hi Byte	Number of Registers - Lo Byte	Checksum CRC - Lo Byte	Checksum CRC - Hi Byte
Byte (Hex)	11	03	00	6B	00	03	76	87

⬆ Read Command

⬇ Read Response

---- Data ----

Packet Comp.	Device Address	Functional Code	Number of Bytes More	Register Value – 1st Byte	Register Value – 2nd Byte	Register Value – 3rd Byte	Register Value – 4th Byte	Register Value – 5th Byte	Register Value – 6th Byte	CRC - Lo Byte	CRC - Hi Byte
Byte (Hex)	11	03	06	AE	41	56	52	43	40	49	AD

Figure 2.10 – Modbus RTU, read holding register command and response packet structure

Next, the write request and response will be discussed. The write command for holding registers has a similar structure to the read command. The first six bytes of the packet structure are the same for the write command as the read command. This contains the information of the device address, function code, address of the first register to index, and the number of registers to be written to the holding registers. The next byte of information in the request is the number of bytes of data that will be contained in the transmission. The next portion is the actual data that is to be written to the 16-bit registers. This data is separated into bytes by the Hi and Lo byte. Finally, the CRC is the last component of the Modbus RTU packet structure for the write command.

After the write command request is sent from the master device, the slave device fulfills the request by storing the data in memory and providing a confirmation that the requested data has been written to the associated holding registers. The write response packet structure is very similar to the packet header information contained in the write command. The first six bytes of the response packet are the same as the first six bytes of the command packet. This information is repeated back to the master

device from the slave device to confirm that the data was written to the holding registers at the requested location. The final component of the write response is the CRC packet component.

Packet Comp.	Device Address	Functional Code	Address of First Register – Hi Byte	Address of First Register - Lo Byte	Number of Registers - Hi Byte	Number of Registers - Lo Byte	Number of Bytes More	Register Value – 1st Byte	Register Value – 2nd Byte	Register Value – 3rd Byte	Register Value – 4th Byte	CRC - Lo Byte	CRC - Hi Byte
Byte (Hex)	11	10	00	01	00	02	04	00	0A	01	02	C6	F0

---- Data ----

Write Command

Write Response

Packet Component	Device Address	Functional Code	Address of First Register - Hi Byte	Address of First Register - Lo Byte	Number of Registers - Hi Byte	Number of Registers - Lo Byte	Checksum CRC - Lo Byte	Checksum CRC - Hi Byte
Byte (Hex)	11	10	00	01	00	02	12	98

Figure 2.11 - Modbus RTU, write holding register command and response packet structure

2.4.4 – Cyclical Redundancy Check (CRC)

The Modbus RTU communication protocol uses a 16-bit cyclical redundancy check (CRC-16) to confirm the integrity and accuracy of the message contents that are transmitted in both request and response messages. The CRC-16 calculation is known for being robust and reliable, with an ability to detect 99.9985% of errors [18]. The CRC calculation follows an algorithm that consists of various logical XOR operations with multiple values and register value bit-shifts. The flowchart for the CRC-16 algorithm can be seen below in Figure 2.12. Each byte of data is sent through this CRC algorithm, and the final value of the “CRC16” variable is the value that is used for the CRC component of the Modbus RTU packet.

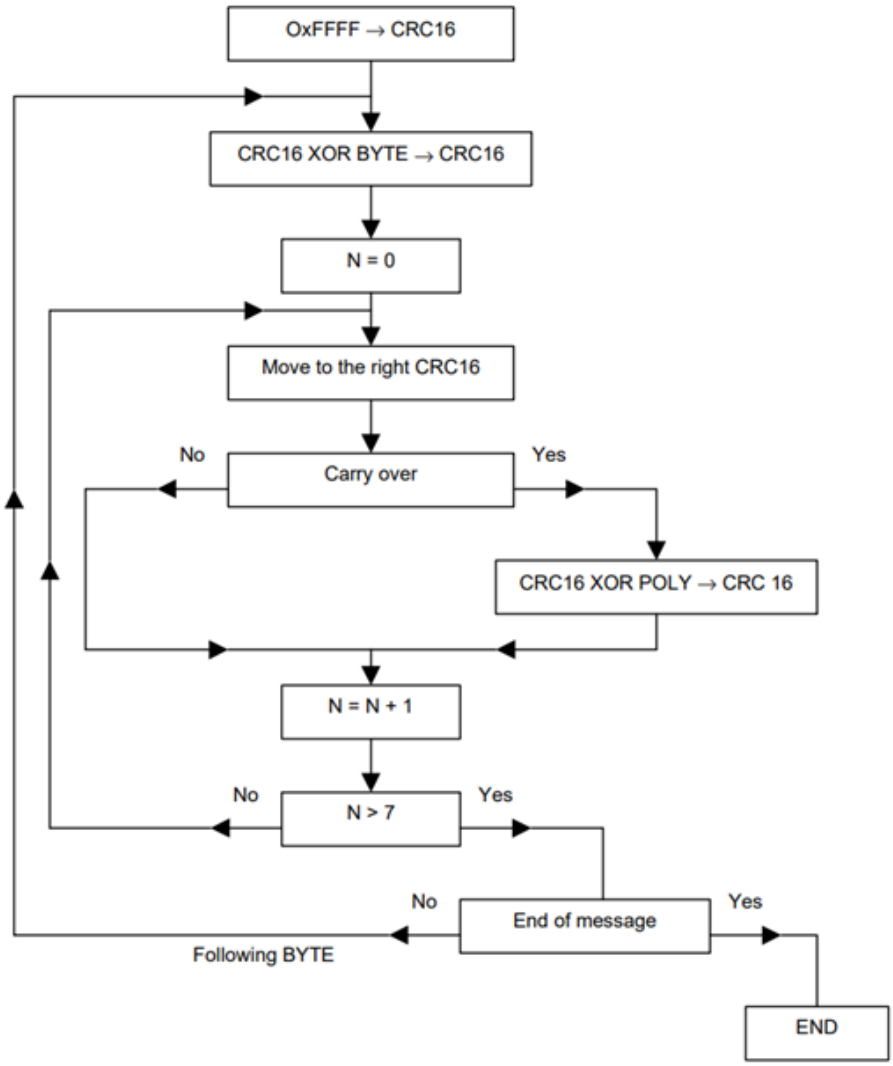


Figure 2.12 - Cyclical Redundancy Check (CRC) flowchart [19]

2.4.5 – Message RTU Framing

The Modbus RTU is an asynchronous communication protocol. As there is not a clock that synchronizes the transmission of messages between a master and slave device, a method for defining the timing of a packet is used. To accomplish this, a frame with a recognized beginning and end point is established. This method allows for the identification of partial messages to be detected, and an error can be reported.

To help define the framing convention of Modbus RTU, a unit of measurement referred to as a “character” is utilized. A character is defined as a 4-bit hexadecimal portion of data, where two characters exist in each 8-bit byte in a message. The time for a character to be transmitted is variable and dependent on the baud rate of the serial transmission. If operating at a baud rate of 9,600 bits per second, a character consisting of four bits will take approximately 416.7µs (microseconds) to transmit. The time values in the following paragraph will assume operation at a baud rate of 9,600 bits per second.

The end of a message is defined as a period of at least 3.5 characters, or 1.4583ms (milliseconds), of silence in the transmission. An example of this silent interval of at least 3.5 characters can be seen below in Figure 2.13. As the data transmission operates in an asynchronous format, the data must be sent sequentially and serially in a continuous stream of characters [20]. If a silent interval of at least 1.5 characters, or 625µs (microseconds), occurs during the transmission, the message is considered incomplete, and the request is not fulfilled by the slave device. An example of acceptable and unacceptable continuous message framing can be seen below in Figure 2.14. This framing structure allows for the error checking of messages before the CRC is verified, and if the message does not conform to the framing standard, then the message is discarded.

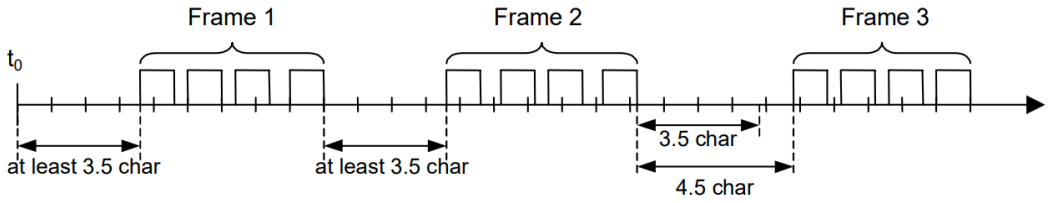


Figure 2.13 – RTU message frame [20]

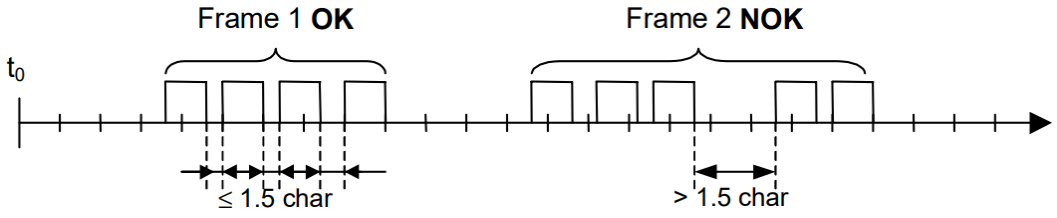


Figure 2.14 – RTU message framing. Acceptable message framing (left). Unacceptable message framing (right) [20].

2.4.6 – Modbus TCP

The Modbus TCP configuration of the Modbus communication protocol allows for the transmission of data through the TCP / IP data transfer protocol [21]. The ethernet interface is used to carry out this communication. How the Modbus TCP packet is structured is similar to Modbus RTU, with an additional header added to the packet to facilitate transmission through the TCP / IP data transfer protocol. As seen below in Figure 2.15, the Modbus RTU packet is stripped of the slave (device) ID and the CRC components, and the Modbus Application Protocol (MBAP) header is added to the Modbus RTU packet. This allows for the transfer of data over the physical ethernet layer.

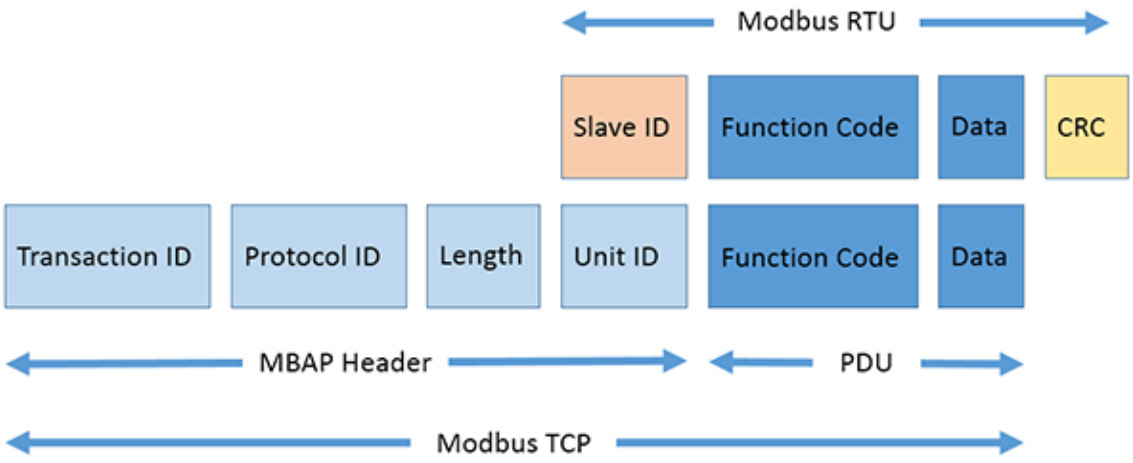


Figure 2.15 – Modbus TCP packet components [21]

2.5 – Power Electronics Unified Controller Board (UCB)

Common architecture Printed Circuit Boards (PCBs) is shared among collaborating researchers. These common architecture PCBs, colloquially referred to as Unified Controller Boards (UCBs) in this paper, provide researchers with a common platform during the design and development phase of projects that promotes rapid prototyping between collaborators. UCBs are intended for multi-purpose use in projects related to topics such as control systems, power electronics, cyber security, and real-time simulation peripherals. To ensure effectiveness in any project, a UCB must be capable of interacting with multiple peripherals through various inputs and outputs of different data types and communication methods. These boards may be responsible for serving as the hardware to operate multiple dependent

and independent processes simultaneously. In this project, the UCB serves as the hardware to host solar inverter control algorithms, facilitates various modular cybersecurity detection and system integrity processes, and acts as the common pathway of communication between interconnected modules [22].

The common architecture and interface of utilizing a UCB among researchers increase the efficiency of the research process. These devices allow for the design flexibility of utilizing multiple topologies within the UCB, as these devices are equipped with a multitude of components and systems to provide a standardized environment. This standardized hardware setup allows for a lower overall platform cost as a new device does not need to be created when transitioning to new or various research projects. These flexible and unified systems also enable researchers to take advantage of code and design reuse to promote rapid prototyping between various developmental phases. The common point of reference provided by a UCB helps facilitate collaboration among researchers in various fields of expertise, such as power electronics, cybersecurity, and control system research.

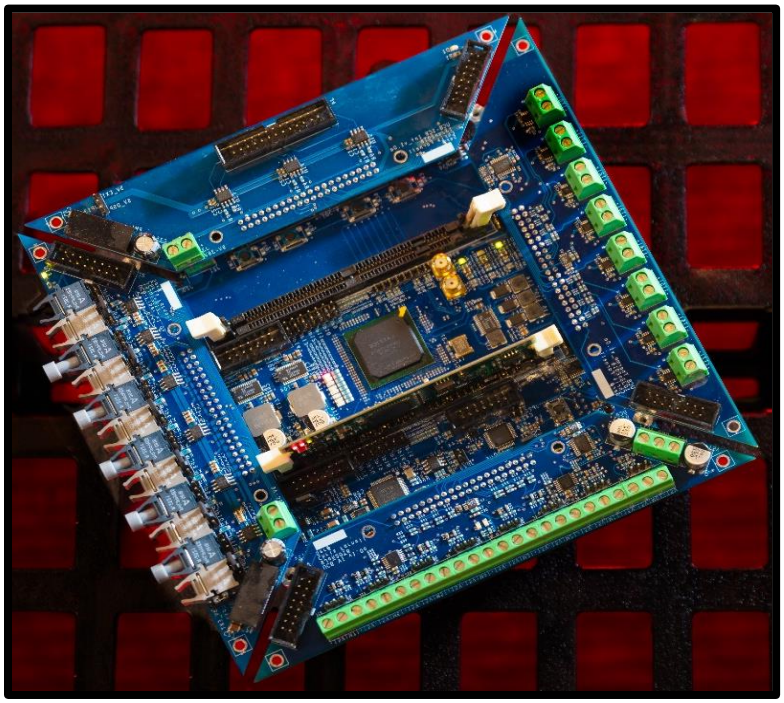


Figure 2.16 – Image of Unified Controller Board (UCB)

2.5.1 – UCB Architecture

The main components within the architecture of the UCB consist of two digital signal processors (DSPs), a field-programmable gate array (FGPA), various communication peripherals and hardware, and input/output (I/O) pins. A diagram of the UCB's architecture can be seen below in Figure 2.17. The UCB contained two single-core Texas Instruments (TI) F28335 Control Card DSPs. The FPGA within the UCB is a Lattice MachX02-7000 model with 6864 lookup tables (LUTs), 240 kb of embedded block RAM (EBR), 256 kb of flash memory, and 484 pins with 334 I/O pins. In addition to the hardware devices, the UCB contains two 8-channel Serial Peripheral Interface (SPI) analog to digital converters (ADCs) and four 40-pin insulation-displacement contact (IDC) connectors. The communication peripherals and components will be discussed in the following subsection.

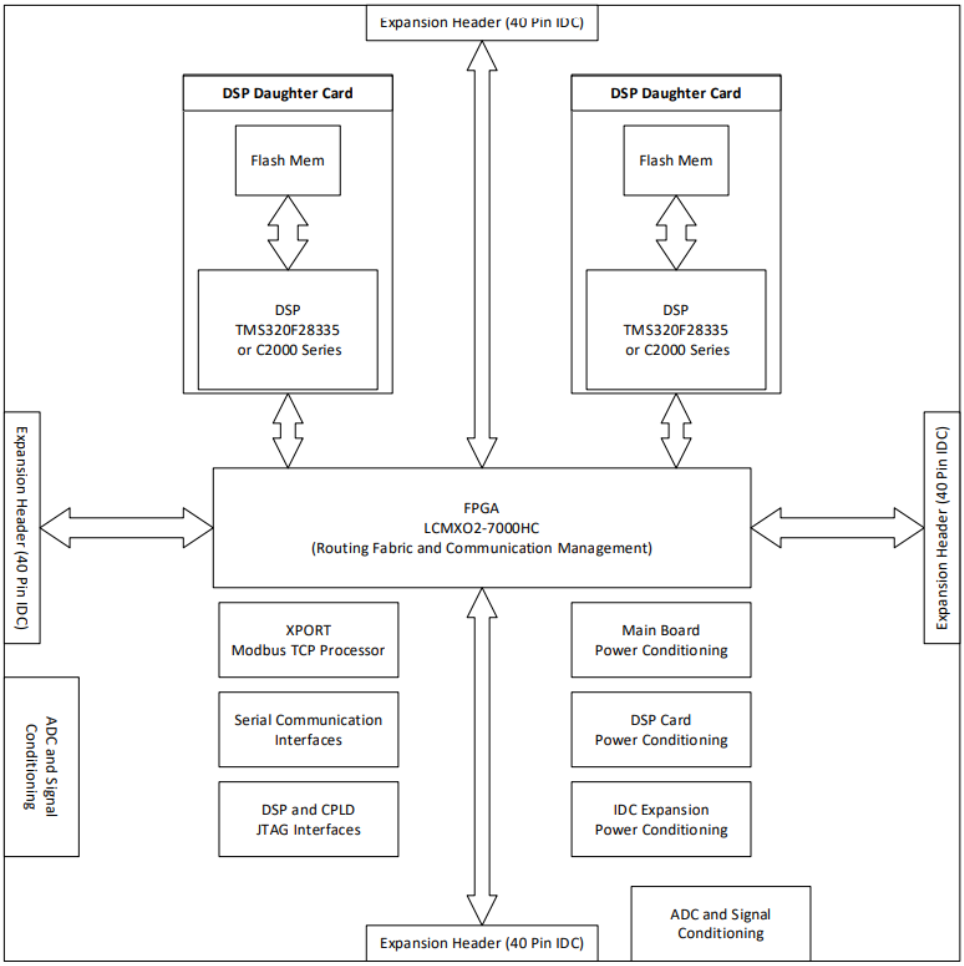


Figure 2.17 – UCB architecture diagram

The UCB interfaces with multiple external systems within the scope of the SETO project. These external systems are LabVIEW, a human-machine interface (HMI) tool, a Raspberry Pi computer, and the power electronics within the SMA inverter. A diagram of these external system connections can be seen below in Figure 2.18. These external connections are used for various components of the multi-level cybersecurity effort of the SETO project and will be discussed in detail in Chapter 3.

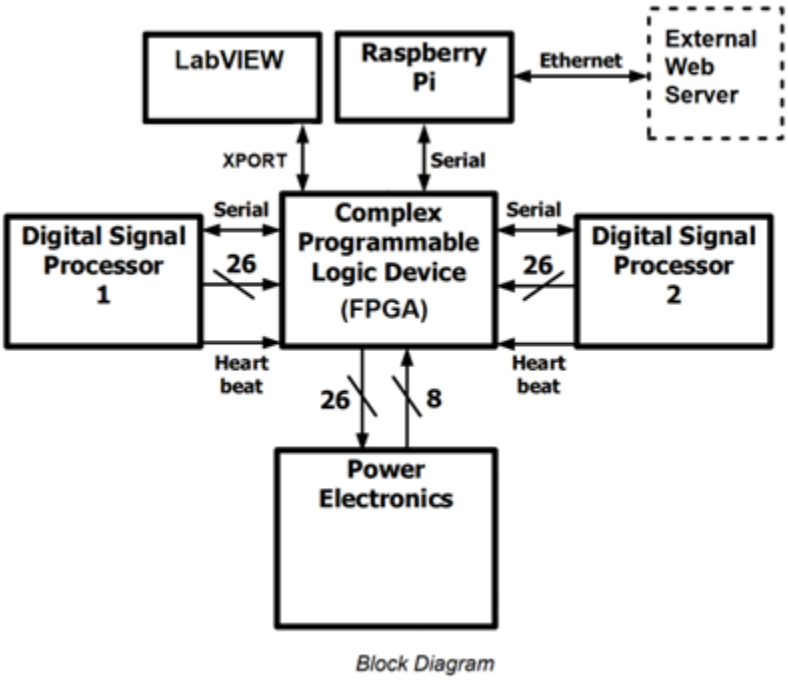


Figure 2.18 – SETO hardware architecture system component block diagram

2.5.2 – Communication Peripherals and Capabilities

The UCB allows for a multitude of peripheral devices to be connected, often in more than one configuration, due to the flexibility of the UCB design. By providing multiple pathways of connection, devices that utilize the same communication hardware can be integrated without the concern of competing resources. In terms of communication, this UCB contains an XPORT Modbus TCP processor and serial communication interfaces such as USB Serial UART via FTDI chipset and an integrated JTAG that can be utilized for serial connections. The FPGA itself can act as the board's routing fabric and communication management, which allows for pinouts to be assigned internally and externally throughout

the board. These pinouts can be utilized to make internal connections with devices such as the DSPs or external connections using the serial communication hardware or the four expansion headers (40-pin IDCs).

The devices that are connected and integrated with the UCB are a Raspberry Pi as a web server, two DSPs hosting solar inverter control algorithms, and LabVIEW acting as the HMI for managing inverter control algorithms and firmware bootloader used by the DSP. The Raspberry Pi is connected through a USB-Mini serial port equipped with an FTDI chip to support serial communications. The DSPs are connected to the FPGA's routing fabric through internal pin sets. LabVIEW is connected through the XPORT connector via Ethernet on the board, which converts Modbus TCP format into Modbus RTU format for serial communications. The connections are highlighted and can be seen in the UCB architecture diagram in Figure 2.19 and the SETO project hardware block diagram in Figure 2.20.

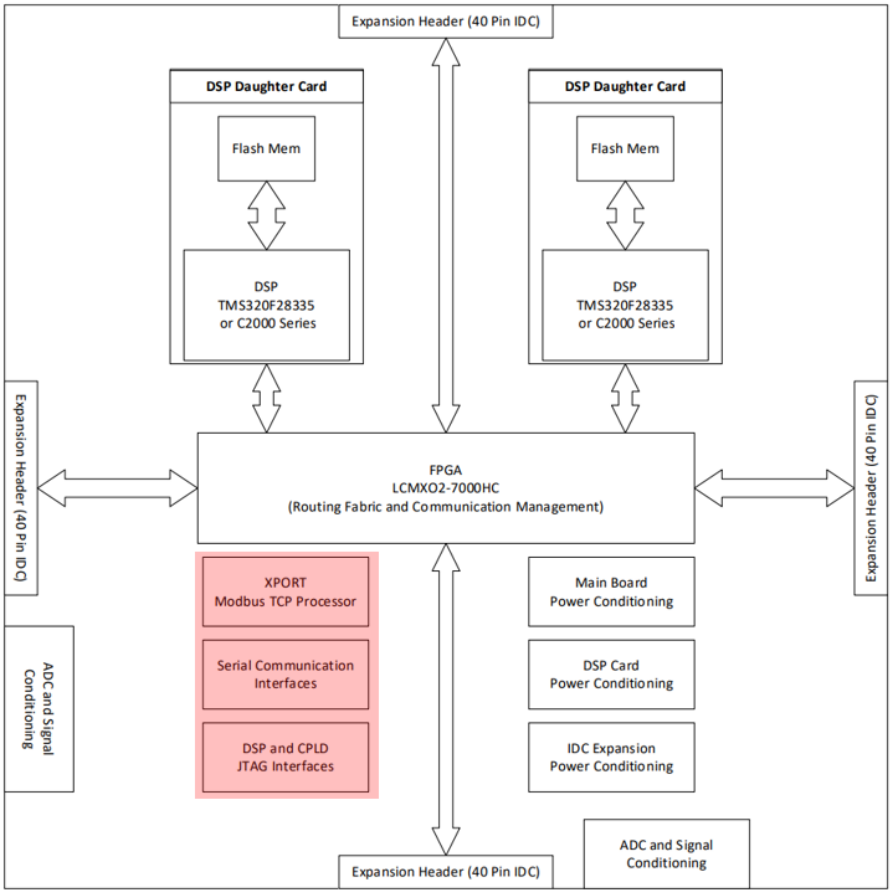


Figure 2.19 – UCB architecture diagram – communication component highlighted

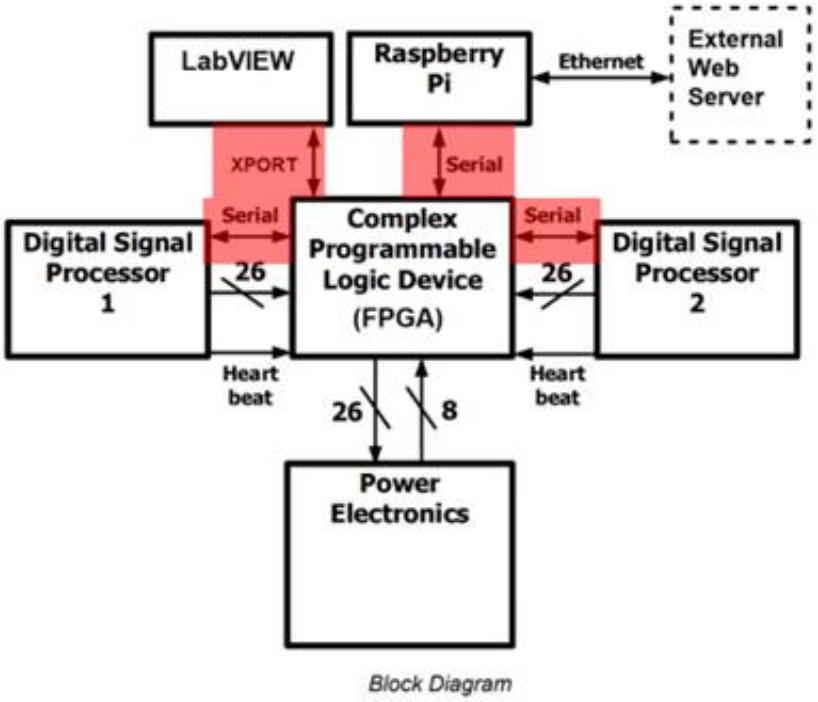


Figure 2.20 – SETO hardware architecture system components block diagram – communication pathways highlighted

2.5.3 – UCB's FPGA

An FPGA is an integrated circuit that can be programmed to implement logical operations through Hardware Descriptions Languages (HDLs) such as Very High-Speed Integrated Circuit Hardware Description Language (VHDL) or Verilog. This highly versatile application of an FPGA allows for the entirety of the UCB to be implemented as an embedded system that is capable of meeting various system architecture needs. The customizable nature of an FPGA allows it to be programmed to facilitate many required logic-based algorithms to implement communication protocols. The UCB contains a Lattice MachX02-7000 FPGA that acts as the central routing fabric for the multiple system components in the board. The location of the FPGA can be seen within the UCB architecture diagram of Figure 2.21 below.

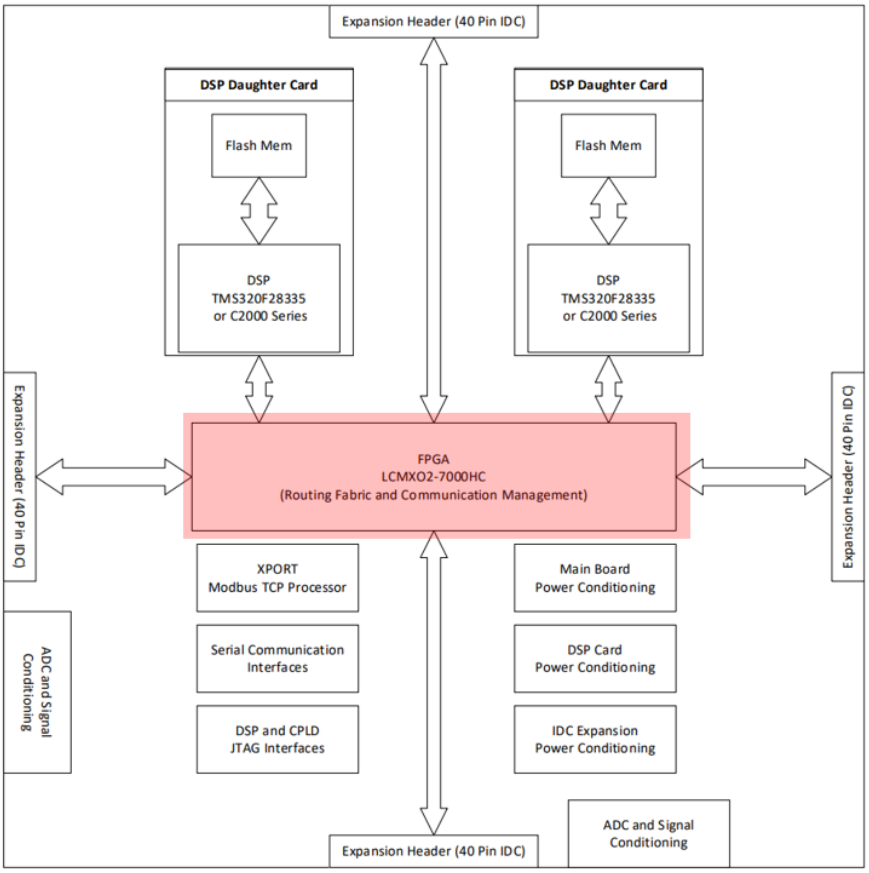


Figure 2.21 – UCB architecture diagram – FPGA highlighted

The specifications of the Lattice MachX02-7000 contain 6,468 look-up tables (LUTs), 240 kb of embedded block RAM, 256 kb of flash memory, and 484 pins with 334 I/O pins. As one of the main goals of the work presented in this thesis is to reduce the required size of an implementation of the Modbus RTU communication protocol, the more pertinent specification of the FPGA to address this goal is that of the LUTs.

2.6 – FPGA Environment and VHDL

An FPGA is an integrated circuit that can be programmed to implement logical operations through Hardware Descriptions Languages (HDLs) such as Very High-Speed Integrated Circuit Hardware Description Language (VHDL) or Verilog. This highly versatile application of an FPGA allows for the entirety of the UCB to be implemented as an embedded system that is capable of meeting various system

architecture needs. The customizable nature of an FPGA allows it to be programmed to facilitate many required logic-based algorithms to implement communication protocols. As VHDL is akin to a low-level language, the use of VHDL within the FPGA allows for these goals and design considerations to be accomplished.

2.6.1 – FPGA Overview

An FPGA is a semiconductor device that utilizes configurable logic blocks (CLBs) in a matrix array, as seen in Figure 2.22, that are connected by programmable interconnects [23]. The nature of FPGA allows them to be reprogrammable indefinitely. This programmability is conducive to the research process, where multiple prototypes of a system may need to be created and tested throughout the design process. The architecture of an FPGA allows for the creation of digital hardware circuits within the CLBs. This allows for designs as simple as basic combinational logic circuits to more complicated designs such as those of IP cores or processor architectures [24].

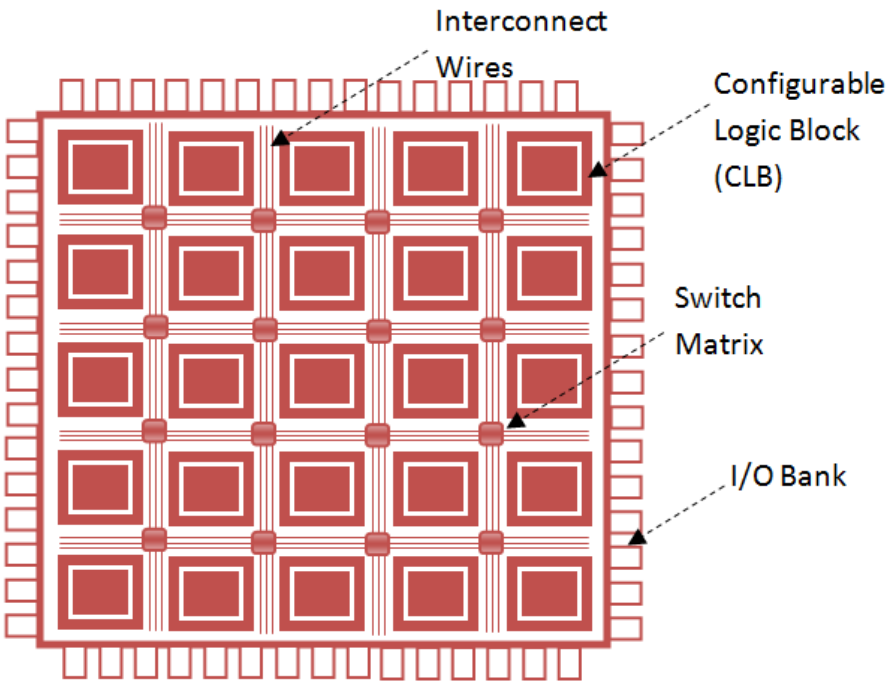


Figure 2.22 – FPGA Architecture with Configurable Logic Blocks [24]

The FPGA used within this research project is the model MachX02-7000 created by Lattice [25], herein referred to as “MachX02”. This specific model of FPGA is also referred to as a Complex Programmable Logic Device (CPLD). That is because this model of FPGA possesses components of both the CPLD and the FPGA. The main application of a CPLD is like that of an FPGA in terms of programmability, though a few distinct differences in their capabilities exist. FPGAs are typically known for being better for applications that are more complex in their design. When programming an FPGA, the program data is stored in volatile memory and is lost upon powering the device off. In contrast, a CPLD is equipped with electrically erasable programmable read-only memory (EEPROM), which allows for faster startup of the device and is non-volatile memory.

Effectively, Lattice has improved the capabilities of the MachX02 by adding EEPROM to their FPGA device. This allows for the FPGA programming to have the attributes of a fast start-up, and the programmed device memory is non-volatile in nature. This allows for a program to persist after a loss of power and assists in the efficiency of the design process and helps facilitate rapid prototyping. Though, the common memory of the device, the RAM, is still volatile in nature. Upon power loss, the stored data in this shared common memory will be lost.

The available resources of the MachX02 are 6,468 Look Up Tables (LUTs), 240 kb of Embedded Block RAM (EBR), 256 kb of flash memory, and 484 total pins with 334 input/output (I/O) pins. The LUTs of an FPGA with respect to this project are akin to the available storage space of a typical personal computer. The input and output pins of the FPGA are what are used as the infrastructure to transfer signals external to the FPGA. The LUTs are the most pertinent measurement of resource utilization with respect to creating the custom Modbus RTU communication protocol.

The LUTs of an FPGA are internal to the architecture of the CLBs, as seen below in Figure 2.23. The structure of each LUT allows for the storing of four bits of data into each LUT and is accessed by a multiplexer (MUX), as seen below in Figure 2.24. When programming an FPGA with the user’s design implementation, in this case, VHDL code, the data for the programming is stored in these LUTs.

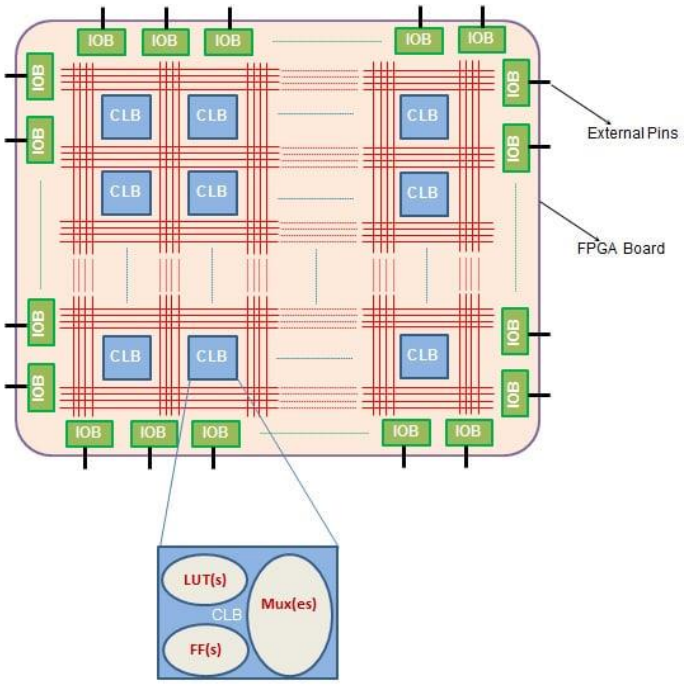


Figure 2.23 – The internal architecture of a typical FPGA [26]

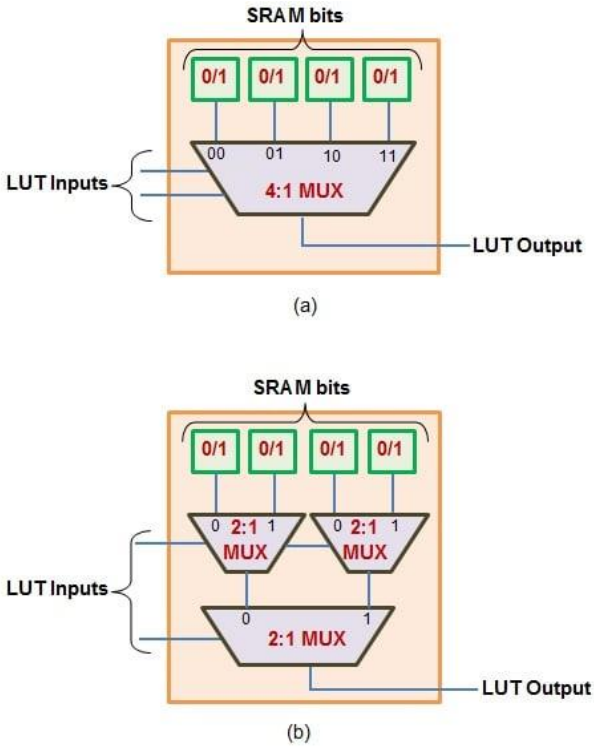


Figure 2.24 – Internal structure of a 4-bit LUT, (a) 4:1 MUX, (b) 2:1 MUX [26]

Before the program data is stored in the LUTs, the user's coding in the hardware description language goes through a process referred to as "synthesis" that converts the text-based code into a netlist. The netlist is a resulting build component that describes the electrical circuit connections that will be formed by the gates and flip-flops of an FPGA. Once the netlist is created, a process referred to as "place-and-route" (P&R) is carried out. The P&R process assigns the netlist elements to the physical components of an FPGA and to the physical resources of the FPGA [27]. This information is then used to create a binary file that is used for programming the user's code into the FPGA.

2.6.2 – VHDL Overview

VHDL stands for "VHSIC Hardwar Description Language," whereas the "VHSIC" portion stands for "Very-High-Speed Integrated Circuit." VHDL is a hardware description language used in digital circuit design that is used to describe hardware models in a human-readable format during the design of digital circuits [28]. Within VHDL, there are three distinct modeling styles, or architectural, styles: dataflow, structural and behavioral [29].

The dataflow style of modeling is used to describe combinational circuits by mapping how data flows through the system [30]. The dataflow style describes typical digital logic hardware that is used in a system design. A typical way of describing this data is through one or concurrent signal assignments. For an arbitrary example, signal "a" and signal "b" may be put through the logical "XOR" operation and assigned to signal "c." If a concurrent signal assignment is made, the resulting signal "c" may then be compared with an additional logical operator of signal "d."

The structural style of modeling defines an entity by describing a set of interconnected components [30]. This is done by connecting various instantiated components of a system to define the functionality of the intended circuit design [28]. This style of modeling is most efficient when a top-level architecture describes the interconnection between lower-level entities.

The behavioral style of modeling describes how a model behaves through algorithms. This is considered the most abstract style of modeling as the code does not directly describe hardware or logical circuit implementation [30]. The behavioral style of modeling utilizes one or more processes that contain an algorithm of sequential statements. These processes can run in parallel, while the statements inside

each process are sequentially carried out. The work presented in this thesis utilizes solely the behavioral architectural modeling style. This is due to the ability to use Next State Logic (NSL), or a state machine, within VHDL.

Within the behavioral style of modeling through VHDL, a syntax exists to create state machine behavior. Next-state logic can be implemented within a process statement by declaring an enumeration type for each state of the state machine [31]. These states are implemented as case statements, where each case is assigned a unique identifier. Within each case statement, an individual may describe an action to be taken for that specific case, such as modifying the values of a variable or performing a gate-level operation. These states can be organized where the conclusion of one case statement leads to a different user-defined case statement. With this architecture, an NSL system can be designed to accomplish a task through with respect to the algorithmic flow of a state machine. An example of a simple state machine is the algorithmic state machine (ASM) that detects the sequence of bits in the order of "1010," seen below in Figure 2.25.

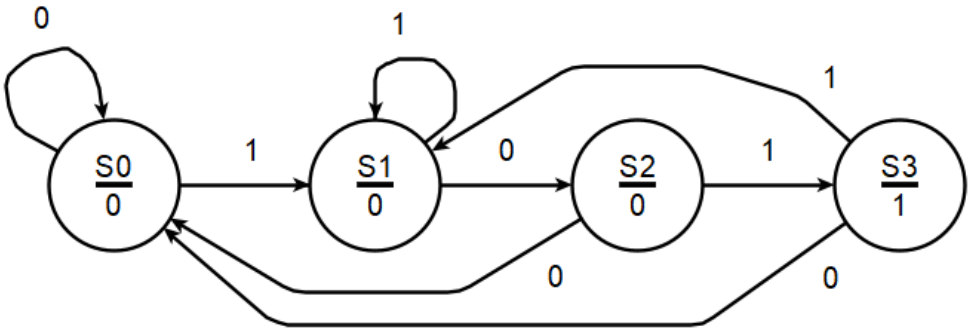


Figure 2.25 – Resetting ASM – Detects bit sequence of "1010"

2.6.3 – Intellectual Property Cores

An Intellectual Property (IP) core is a reusable functional block of logic are data that can be used within an FPGA used for a specific application [32]. IP cores are typically a singular module that is capable of operation on its own and is part of a larger system or design. The modular format of the IP core often allows it to be utilized in various applications where the specific functionality of that IP core

may be needed. One example of an application of an IP core is to implement a processor architecture within the hardware environment of an FPGA. This processor architecture allows for some aspects of a conventional computational processor. One common application is utilizing a high-level programming language, such as C, to be used in an FPGA-based project. As the IP of the IP core stands for intellectual property, it is common that these modular logic blocks require licensing rights to be observed when used. The licensing availability of IP cores ranges from strictly prohibited, requiring explicit permission, to open-source licensing and availability.

An IP core resides within the CLBs of an FPGA and requires use of the available resources of the FPGA to function. As the IP core is a physical digital logic representation of a system, the FPGA that houses the IP core must have enough resources to implement the IP core and any additional code that will be utilizing the IP core. The IP core is essentially a component of the final code-based implementation within an FPGA that operates in conjunction with any additional user components. A diagram depicting an IP core residing within an FPGA with additional digital logic code-based components of the user can be seen in Figure 2.26. An example of this IP core (seen in blue) may be an embedded processor architecture that is connected to the user's components (seen in green) that work in conjunction to complete a specified task or implement a system.

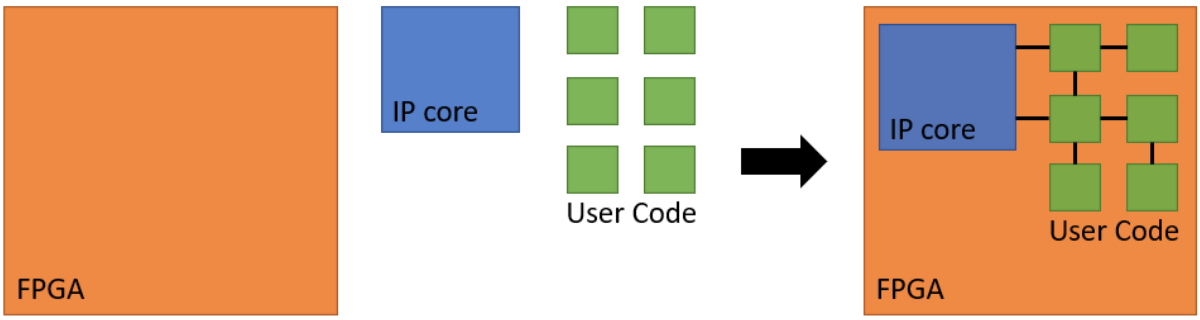


Figure 2.26 – Diagram of IP core and user code components using the available resources of an FPGA; Illustrates an IP core resides with user modules

Chapter 3

Modbus RTU FPGA Implementation via VHDL

3.1 – Introduction

As UCBs are capable of a wide variety of applications and often host multiple integrated systems, the efficient use of physical and memory space is of great concern. To ensure that a project has access to the required system resources for each aspect of its design, software optimization is often necessary during the design process. Concerning the custom implementation of the Modbus RTU communication protocol presented in this thesis, strong consideration for conserving available resources can be achieved through coding the communication protocol with VHDL. While the most significant design concern of this project regards the minimization of resource utilization, the second most significant concern is that of providing a standardized communication method between modules for this SETO project to facilitate efficient research collaboration.

The standardized format of Modbus RTU, coupled with its wide use and reliable method of transmitting data, makes it a strong candidate when selecting a standardized communication protocol. As mentioned previously, Modbus RTU is widely used in ICS and is known for being reliable and relatively straightforward to implement. The modular components that implement various cybersecurity-focused applications into the solar inverter controller of the SETO project are all capable of communicating internally with the FPGA via Modbus RTU. The SMA solar PV inverter model also supports the Modbus RTU communication protocol. This communication protocol, coupled with the previously mentioned favorable aspects of Modbus RTU, justifies its use as the main mode of data transmission within the SETO project.

3.2 – System Components

This subsection covers the system components of the Modbus RTU communication protocol implementation and the respective modular components of the greater SETO project that require data transmission through this protocol. Section 3.2.1 discusses the modular components of the SETO project that require Modbus RTU communication, along with a summary of their application.

3.2.1 – SETO Modules Requiring Communication

The UCB utilized in this project contains all the hardware required to implement the Modbus RTU protocol and interface with the various cybersecurity-focused modular components of the SETO project. The architecture of the SETO project concerning Modbus RTU data transmission can be categorized into three main hardware components with an additional external human-machine interface (HMI). These components are the FPGA, the two DSPs, and a Raspberry Pi (RPi). A diagram of the interconnects of this architecture can be seen in Figure 3.27 below, also pictured with an image of the UCB and an image of the UCB connected to a Raspberry Pi and non-SETO project-related power electronics board.

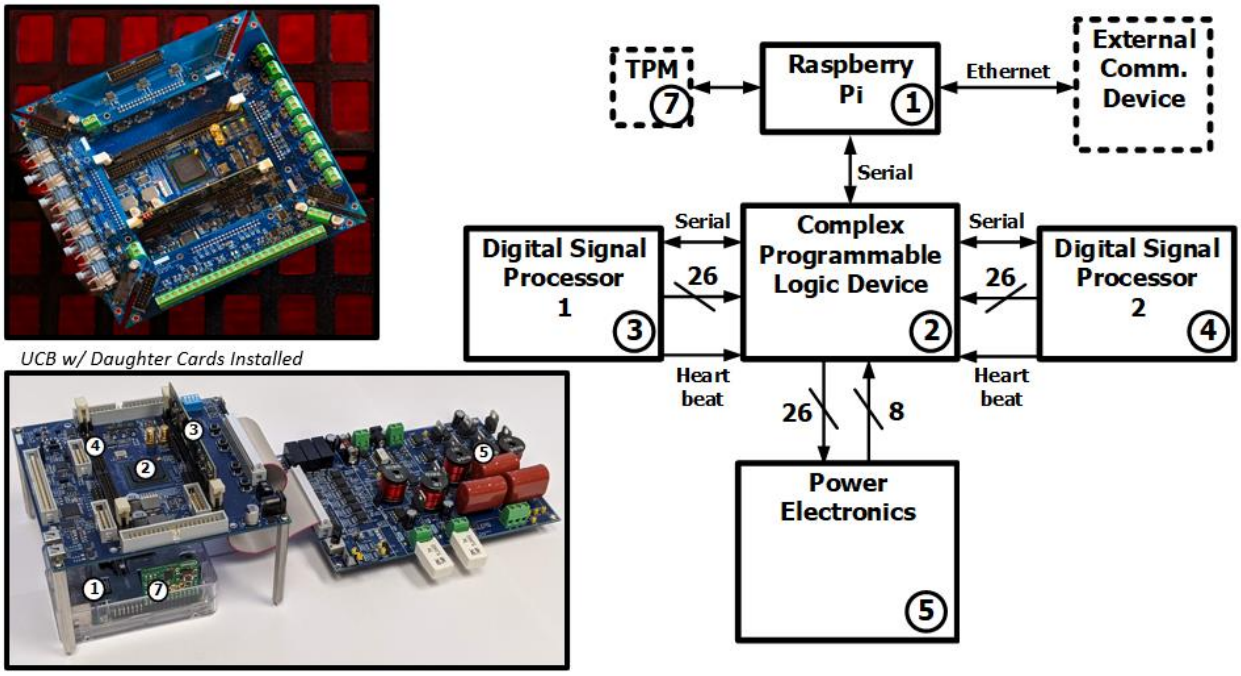


Figure 3.27 – Diagram of SETO project architecture interconnects (right), UCB (top left), and UCB connected to Raspberry Pi and power electronics board (bottom left)

These three major components of the SETO project are connected via the Modbus RTU implementation within the FPGA. The Modbus RTU within the FPGA is the routing fabric for all communication and data transmission within the SETO project. The FPGA is the link between the DSPs and RPi that provides access to the shared common memory register bank within the FPGA. This shared register memory is the central point for all data that may need to be accessed by multiple modules within

the SETO project. Additionally, the UCB supports a Modbus TCP connection through local ethernet, which allows for a remote connection to this system. A network connection diagram of these interconnections can be seen below in Figure 3.28.

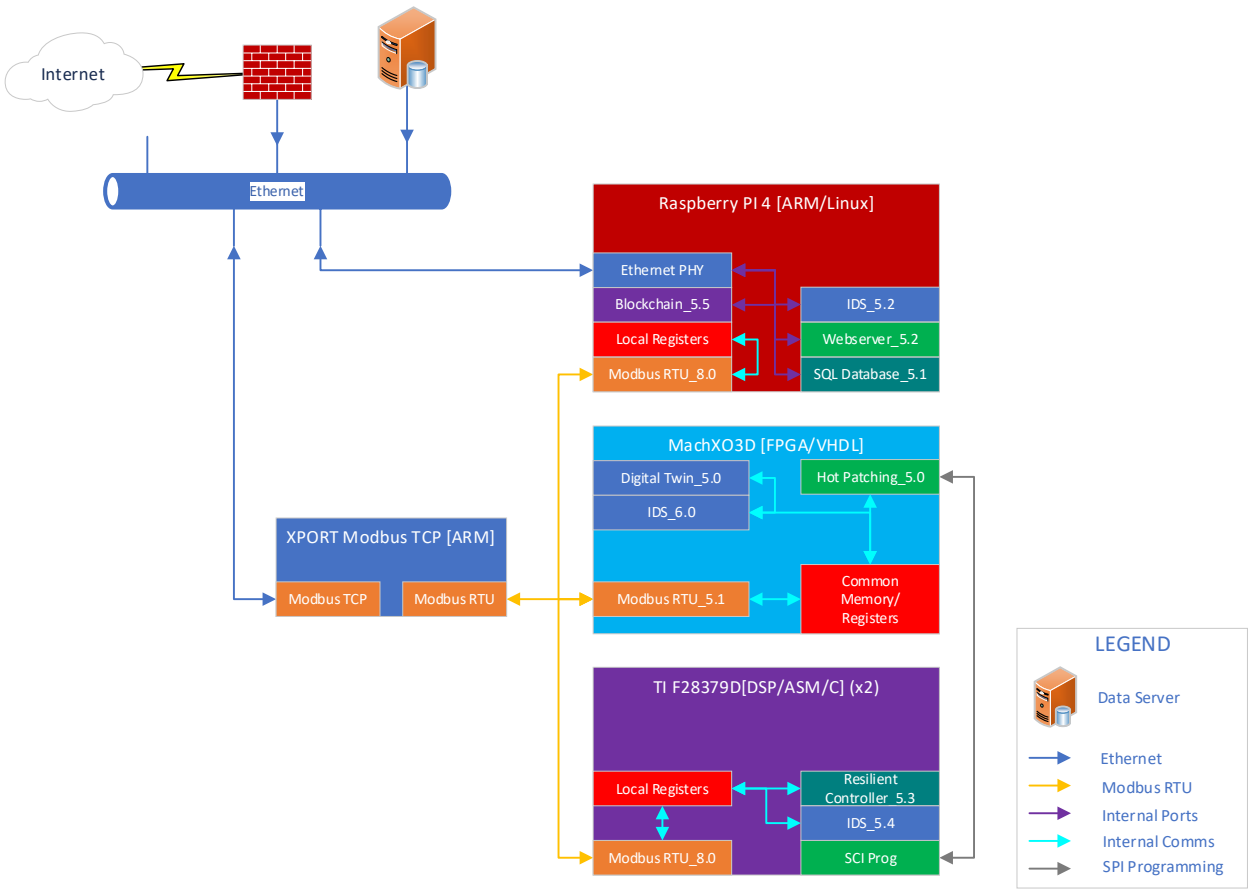


Figure 3.28 – SETO module network interconnection diagram of FPGA, DSPs, Raspberry Pi, and ethernet/internet (numbers after module name denote SETO task identifiers)

FPGA

The Modbus RTU implementation within the FPGA is the component of the SETO project that acts as the central routing fabric for all communication between modules and can be seen in the hardware architecture system component block diagram below in Figure 3.29. In addition to containing the Modbus RTU implementation, the FPGA contains three additional components of the SETO project that require access to data transmission.

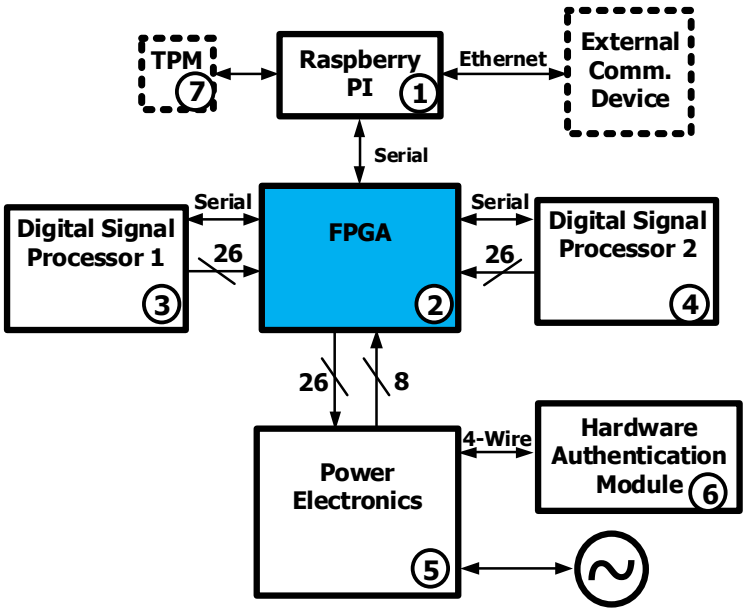


Figure 3.29 - SETO hardware architecture system component block diagram, FPGA highlighted

The three additional modular components of the SETO project operated within the FPGA are the firmware bootloader, Digital Twin, and hardware-level protection features. The firmware bootloader is a VHDL component responsible for downloading a binary bit file of inverter control algorithms into the FPGA, used within the Digital Twin architecture. Additionally, this firmware bootloader is used to update or load new inverter controller firmware into the DSPs. The Digital Twin (DT) architecture within the FPGA can emulate the ANPC inverter controller algorithm of the SMA inverter. This system performs various system checks on the firmware to verify its integrity, safety, and authenticity. Additionally, this DT component can orchestrate the hotpatching between inverter controller firmware seamlessly between two DSPs in real-time. A diagram of the DT within the UCB and its relation to the Modbus RTU communication protocol can be seen below in Figure 3.30. The diagram depicts the DT residing within the UCB and the connection between the DSPs, the FPGA acting as the routing fabric, and the Modbus RTU communication protocol implementation. Additionally, a Modbus TCP interconnect can be seen using the XPORT adapter. This adapter within the UCB can convert Modbus RTU packets to Modbus TCP packets and vice versa to utilize ethernet network connection capabilities.

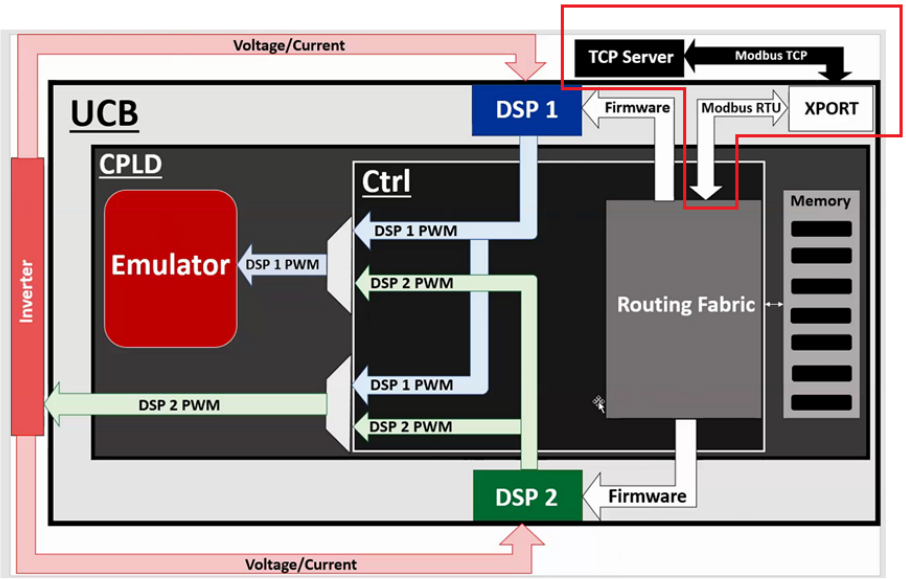


Figure 3.30 – Diagram of Digital Twin within UCB of SETO project, Modbus RTU communication components outlined in red (top right)

Raspberry Pi

The RPi contains four components of the SETO project and is connected to the FPGA through a Modbus RTU connection, which can be seen in the hardware architecture system component block diagram below in Figure 3.31. These four processes are a web server, Snort Intrusion Detection System (IDS), a blockchain implementation, and a Device-Level IDS. The web server operates the collection of historical operational data of the solar inverter and provides a human-machine interface (HMI) for the user to access operational status information. The Snort IDS operates as a network-based IDS to mitigate any cyberattacks from the network external to the UCB and Raspberry Pi. The blockchain application manages the integrity and handling of various file systems within the SETO project. Finally, the device-level IDS mitigates unwanted access to the device at a system level. All these components rely on data that is held within the shared memory registers residing within the FPGA and is accessed through the Modbus RTU communication protocol.

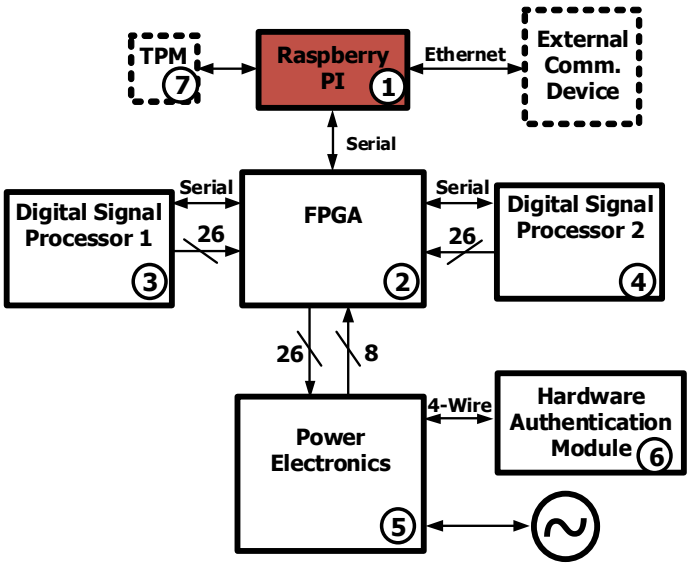


Figure 3.31 - SETO hardware architecture system component block diagram, Raspberry Pi highlighted

DSP

The DSP contains three components of the SETO project and is connected to the FPGA through a Modbus RTU connection that can be seen in the hardware architecture system component block diagram below in Figure 3.32. These three components are the primary resilient controls, inverter communications, and the device-level IDS. The primary resilient controls control the SMA inverter with an algorithm resistant to various detrimental cyber-physical events. Internal communications are responsible for sending and retrieving data from the shared memory of the FPGA used within the inverter controller algorithm. Finally, the device-level IDS is another portion of the security component we saw earlier in the Raspberry Pi. Additionally, the DSPs contain their unique flash memory used for various communication and inverter controller functions. The DSPs can be seen in the UCB architecture diagram below in Figure 3.33.

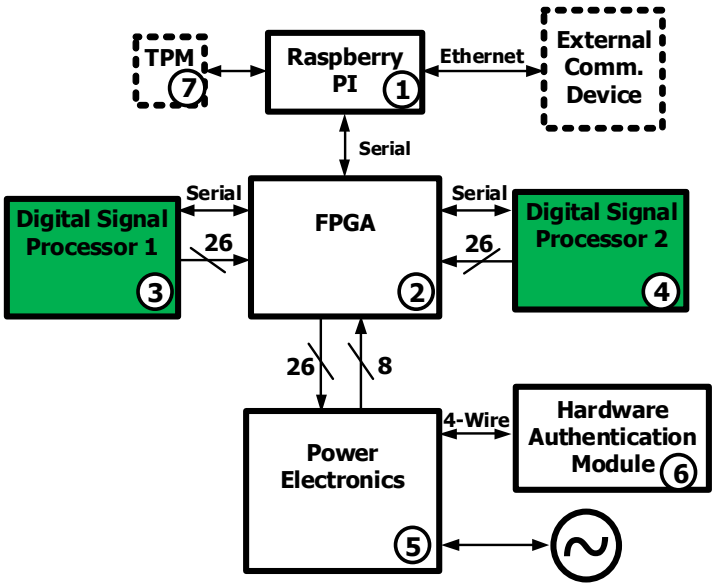


Figure 3.32 - SETO hardware architecture system component block diagram, DSPs highlighted

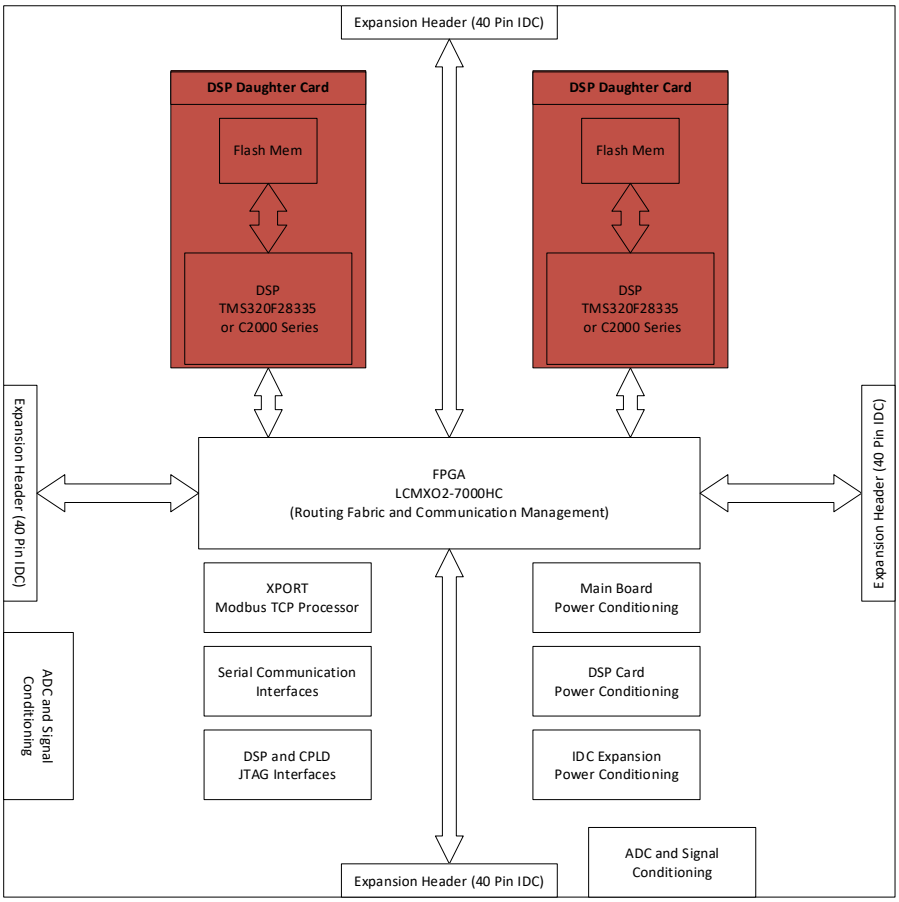


Figure 3.33 - UCB architecture diagram – DSPs highlighted

LabVIEW

The various HMI setups used in the design process and final implementation of this project was created with the software LabVIEW. LabVIEW and the FPGA are communicated through a Modbus TCP connection converted from the Modbus RTU format through the XPORT module on the UCB. During the design and development of this Modbus RTU implementation, various LabVIEW HMIs were used to test and troubleshoot the embedded Modbus RTU implementation. The final LabVIEW HMI facilitates three processes regarding the SETO project. These processes are the bootloading of inverter controller firmware, the firmware integrity and validation modules using the DT, and the interface to facilitate hotpatching between DSPs. The application and use of the various LabVIEW HMIs will be discussed in further detail in Chapter 4, Test Setup and Experimental Results.

3.2.2 – Communication and Bus Infrastructure

The serial bus infrastructure for this Modbus RTU implementation within an embedded system is based on a modified communication and bus infrastructure project by Dr. Chris Farnell with the National Center for Reliable Electric Power Transmission (NCREPT) at the University of Arkansas. This communication and bus infrastructure created a basic custom serial communication protocol to send and receive data between power electronic systems and LabVIEW HMIs. This basic serial communication protocol was designed by creating a state machine using next-state logic (NSL) with the behavioral modeling style of VHDL within an FPGA. To create the custom Modbus RTU implementation, this prior existing communication and bus infrastructure was utilized, and a new NSL algorithm was developed.

The main components of the communication and bus infrastructure are the bus interface, bus master, first-in-first-out (FIFO) registers, local memory, and the shared common memory. The bus interface and bus master are VHDL component files that describe their respective hardware architecture. The FIFOs, local memory, and shared common memory are architectural resources in the form of registers. These components form the infrastructure to facilitate the NSL algorithm of the Modbus RTU VHDL implementation and manage the asynchronous serial flow of signals during transmission.

Bus Interface

The bus interface architecture component is the VHDL implementation that describes the hardware infrastructure to support the signal transmissions between components. The architecture of this component is the Common Bus Architecture (CBA) developed by Texas Instruments. The CBA was created as a scalable and convenient method to connect logical blocks within a system-on-chip (SoC), such as an FPGA [33]. As the bus interface VHDL module was created based on the CBA communication infrastructure, it can be instantiated multiple times. As this method of implementation is scalable, these instantiations each have their signal pathway assigned to their designated resources. The serial bus instantiations internal to the FPGA wave signal pathways are routed to the common memory, as seen below in Figure 3.34, to give the various SETO module components access to the shared common memory. These internal signals share a common serial communication bus. The internal signals connected to the common memory are connected to the external communication peripherals through the Modbus RTU NSL algorithm that prepares the individual data components for transmission.

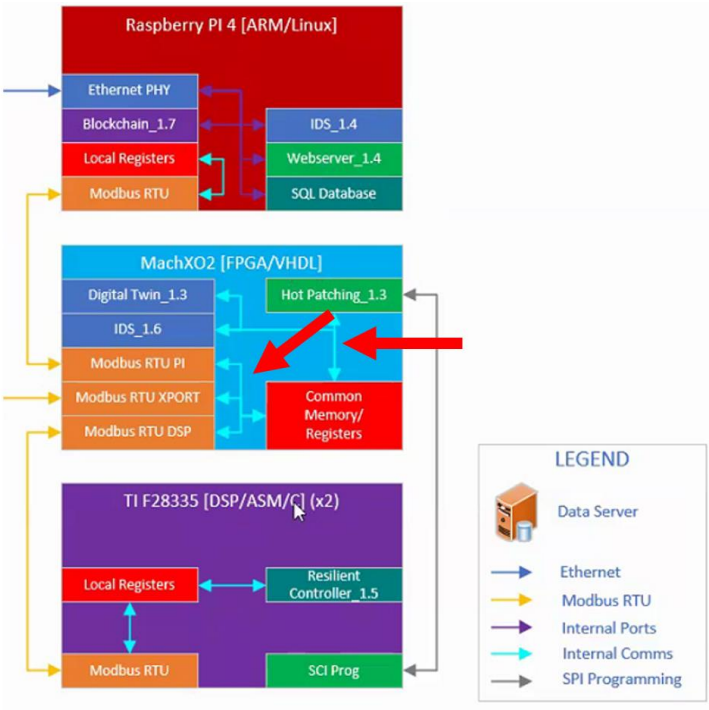


Figure 3.34 - SETO module network interconnection diagram of FPGA, DSPs, Raspberry Pi, and ethernet/internet. Red arrows indicate the internal serial bus infrastructure of the Bus Interface.

Bus Master

The bus master architecture component is the VHDL implementation that describes the hardware infrastructure that manages access to the common serial communication bus internal to the FPGA. As this is a shared communication resource with multiple Modbus RTU module instantiations, the use of a bus master to prevent bus contention from occurring is employed. The bus master module establishes and manages the common bus through three components or signals. The first set of corresponding signals manages when access to the common bus is requested and when access has been granted. These corresponding signals handle requests for bus control and ensure that only one device is utilizing the common bus at one time. Additionally, a component ensures data is available when a bus request is asserted. This addresses the potential of corrupted or unavailable data gaining access to the common bus. The final considerable component of the bus master is the priority encoder. As different modules within the UCB and SETO project have varying degrees of critical system impact, it is imperative to implement a system to manage communication access priorities. A priority level is assigned to each component within the FPGA to ensure the most critical components of the greater project have prioritized access to the common bus to carry out their data transmissions.

First-In-First-Out Registers (Read and Write)

One of the major components that allow for the full-duplex transmission and instantiation of multiple Modbus RTU modules is the FIFO registers. Each Modbus RTU module contains a set of two FIFO registers, one for incoming communication and one for outgoing communication. These register components are paramount to the throughput of the Modbus RTU communication. In the event multiple packet transmissions are in queue to have access to the common bus, the messages can still be handled. All incoming and outgoing transmissions are first stored in these FIFO registers before being processed by the NSL algorithm.

The Modbus RTU NSL algorithm utilizes these FIFO registers for incoming and outgoing data. If the data is coming into the Modbus RTU module, the NSL algorithm handles the packet components one byte at a time using the FIFO register. These bytes are popped out of the FIFO register and handled by their respective case statements of the NSL algorithm for each transmission. If multiple packet

transmissions are in this FIFO, the Modbus RTU packet components contain enough information to determine the beginning and end of a packet's contents. If a packet component is incomplete or inaccurate, error handling can discard the packet and respond accordingly. Like the incoming data, the Modbus RTU module handles the packet components of outgoing data one byte at a time.

Local Memory (RAM)

Each Modbus RTU module implementation contains its unique and local personal memory in the form of RAM. This memory access facilitates the packet component processing of the Modbus RTU NSL algorithm. All the incoming and outgoing bytes of data are stored in their unique memory registers. These values stored in the memory registers are then used for the Modbus RTU algorithm and the CRC calculation. As an incoming message is decoded or an outgoing message is formed, all packet components pass through these memory registers and are stored and utilized by the Modbus RTU NSL algorithm.

Shared Common Memory

The shared common memory of the FPGA acts as the common holding area of all UCB-based data required by SETO project modules. This shared common memory, in conjunction with the FPGA, acts as the routing fabric for all data flow within the UCB. When a read or write command is sent to one of the Modbus RTU modules, all data is stored in or read from this shared common memory. Each register in this shared common memory is 16-bits wide.

3.3 – Modbus RTU Design

This section will discuss the design process of the embedded Modbus RTU design while discussing the parameters and components of the NSL algorithm in detail. Subsection 3.3.1 will discuss the design parameters considered during the development process. Subsection 3.3.2 will overview the NSL algorithm used to create this embedded Modbus RTU implementation. Subsection 3.3.3 discusses the processing of incoming Modbus RTU commands, while subsection 3.3.4 elaborates on how these command requests are fulfilled and the corresponding response message. Finally, the CRC algorithms

are discussed in subsection 3.3.5, while subsection 3.3.6 will cover the error handling in case of a CRC validation failure.

3.3.1 – Design Parameters

This custom implementation of the embedded Modbus RTU functionality must follow specific standardized communication protocol parameters. As a primary goal of this thesis work is to improve the efficiency and effectiveness of researchers by providing a standardized method of data transmission among multiple interfaces, it is imperative that the design of this Modbus RTU implementation conform to the required standards.

RS-232

The most basic standardization about serial communication to conform to is the RS-232 format. This recommended standard defines the voltage levels a signal uses to determine if that signal is asserting a binary ‘0’ or ‘1’. A signal is valid under RS-232 when it is in the range of +3 V to +15 V or the range of -3 V to -15 V. A voltage signal is characterized as asserting a binary ‘0’ if the signal voltage lies in the defined positive range and asserting a binary ‘1’ if the signal voltage lies in the defined negative range [34]. This voltage level is essential to consider as the signal voltage levels sent to the peripheral communication devices of the UCB need to be capable of meeting this standard. As the FPGA of this work supports positive and negative voltage levels of 3.3 V, this device can meet the signal level voltage standard of the RS-232 format.

Full Duplex

Another design consideration to conform to the RS-232 format is the transfer type used to send and receive messages. RS-232 can operate as a full duplex system, which utilizes two signal pathways among four connection points. This provides dedicated transmission and receiving bus line connections for the system. The full duplex infrastructure allows for the simultaneous sending and receiving of messages. This is achieved by connecting the transmission signal output of the transmitting device to the receiving signal input of the receiving device and the transmission signal output of the receiving device into the receiving signal input of the transmission device, as seen below in Figure 3.35.

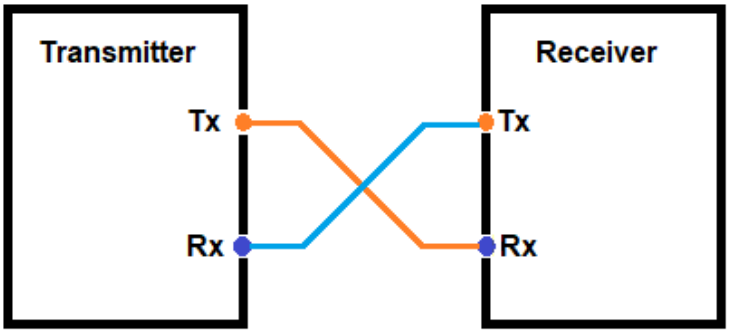


Figure 3.35 – Full duplex signal transmission connections

Baud Rate

The speed at which bits are serially transmitted within the Modbus RTU communication protocol is customizable and explicitly follows a strict standard. Though, common baud rates for transmission are typically used. The baud rate used for this implementation of Modbus RTU operates at a baud rate of 9,600 bits per second. This baud rate puts the transmission of each bit of data at one ninety-six hundredth of a second, or 104.167μs per bit.

Modbus RTU Framing

As Modbus RTU is an asynchronous serial communication protocol, it does not utilize an external clock to manage the beginning and end of packet transmission. To account for this, a framing method for Modbus RTU defines the beginning and end of messages. This is done by defining parameters for durations of silence on the serial line that indicate when a message transmission has ended. The unit to measure these durations of silence is referred to as a character. Each character consists of 11 bits: a start bit, 8 bits of data, a parity bit (or an additional stop bit if no parity is implemented), and a stop bit [20]. As the baud rate is configured as a variable parameter, the baud rate chosen for the system dictates the length of time of one character. The chosen baud rate for this design is 9,600 bits-per-second and equates to transmitting one bit of data at 104.167μs per bit. The end of a message frame is dictated by a duration of silence of at least 3.5 characters, while a message is considered invalid if a silence of 1.5 characters occurs in the middle of the message frame. Table 2 below shows the character time durations used within the design process concerning framing messages.

Table 2 – Modbus RTU Framing Time Measurements

Value (at 9600 baud rate)	Duration (time)
Single bit	104.167µs
1.5 Characters	1718.766µs
3.5 Characters	4010.430µs

To achieve a minimum delay of 3.5 characters between transmissions, a duration of silence of 3.675 characters was targeted. This 3.675 characters is 0.175 characters over the required minimum silence (approximately 4.7%) and provides adequate room for unforeseen errors in timing. The approximate time for 3.675 characters to pass at a baud rate of 9,600 bits-per-second is 4200µs. As the Modbus RTU VHDL implementation uses a next-state logic algorithm (covered in the following sections), a calculation concerning the period of the clock of FPGA is made. The clock of the FPGA operates at a frequency of 25 Mhz. To properly frame the Modbus RTU message, a counter is incremented to a specified number before putting the system into a state that is ready to send or receive the next message. This counter requires two clock cycles per increment. The chosen value to increment to that indicates a period of 3.675 characters (4200µs) has passed is 52,500. The calculation for this value can be seen below.

$$25\ Mhz\ \cdot 4200\mu s = 105{,}000\ \text{clock cycles} \rightarrow \frac{105{,}000\ clock\ cycles}{2\ \frac{clock\ cycles}{increment}} = \mathbf{52{,}500}\ \boldsymbol{increments}$$

3.3.2 – Next State Logic Algorithm - Overview

The core of the custom Modbus RTU implementation is carried out through the behavioral architecture modeling style of VHDL. As mentioned in the background section, this modeling style allows for using next-state logic (NSL) algorithms. These NSL algorithms carry out lines of code sequentially based on case statements within the defined process of the VHDL code. This allows for a state machine to be created that can carry out the functions of the Modbus RTU communication protocol within an embedded system, such as an FPGA.

An example via code snippet of case statements being used to implement the Modbus RTU communication protocol can be seen below in Figure 3.36. In this figure, the declaration of the beginning of the case statement process can be seen on line 737, where the code “case CS_FIFO_Bus is” is present. Next, on lines 739 and 751, the term “when S0=>” and “when S1=>” are present. These lines of code denote the beginning of a unique case statement with the identifiers of “State 0” and “State 1”, respectively. The contents of S0 consist of a perpetual loop that checks to see if a command is being received into the FIFO, along with asserting the reset state for all pertinent register values. This case state is both the reset state and the beginning of the receiving command process. The code content of S1 begins the first steps of processing the received command. It stores the first incoming data byte in the FPGA’s local RAM at the hexadecimal address “0x00” for later use when fulfilling the requested read or write command.

```
737         case CS_FIFO_Bus is
738
739             when S0=>
740                 if(STD_FIFO_R_Empty = '1') then        --Check to see if commands are in queue
741                     NS_FIFO_Bus<=S0;
742                 else
743                     NS_FIFO_Bus<=S1;
744                     STD_FIFO_R_ReadEn <= '1';          --Assert Read Signal for FIFO
745                 end if;
746                 Rcv_Cnt_rst<='0';
747                 Buf_Cnt_rst<='0';
748                 Reg_Cnt_rst<='0';
749                 RAM_Cnt_rst<='0';
750
751             when S1=>                                  --Read Command from FIFO
752                 Temp_Cmd<=STD_FIFO_R_DataOut;          -- Start Delimiter (OG)
753                 LD_Temp_Cmd<='1';
754                 RAM_Data_In <= STD_FIFO_R_DataOut;     -- Device Address, Ram Address 0X0000
755                 RAM_address <= X"00";
756                 RAM_wea <='1';
757
758                 NS_FIFO_Bus<=S2;
```

Figure 3.36 – Case statements of next state logic algorithm with VHDL behavioral architectural modeling style

The NSL algorithm of the Modbus RTU implementation contains approximately 250 unique case states to carry out the communication protocol. These unique case states can be sorted into four categories: command request processing, command response processing, CRC calculation or verification, and error handling. Each category will be discussed in detail in their subsection below.

The state diagram of the high-level overview for the NSL algorithm can be seen below in Figure 3.37. The basic pathway for the NSL upon the slave device receiving a request from the master device is to determine if the function is a read or write command. Next, the packet values are stored in the local RAM for processing. Once the CRC calculation is verified to confirm transmission integrity, the requested action is performed, and the appropriate response packet is formed and sent back to the master device.

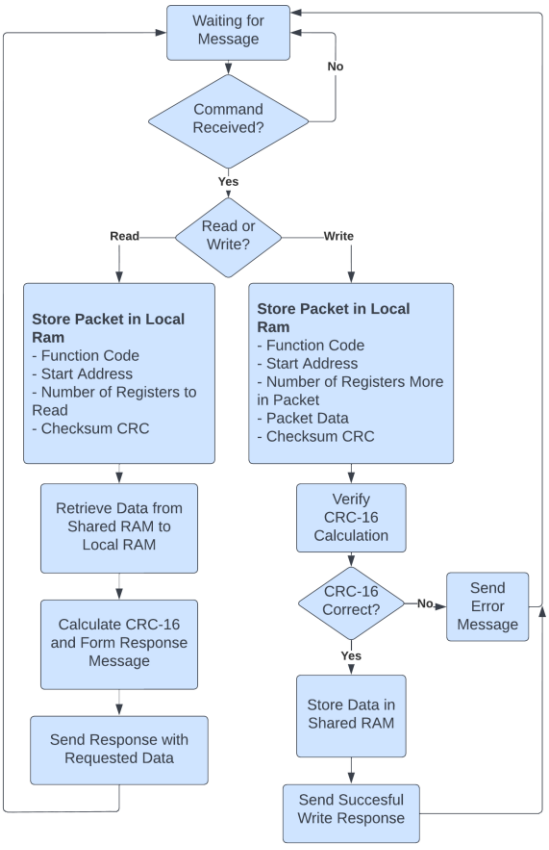


Figure 3.37 – Overview of the state diagram of the Next-State-Logic algorithm for the Modbus RTU VHDL implementation

3.3.3 – Command Request Processing Algorithm

This subsection will discuss the portion of the NSL algorithm that processes the request commands within the Modbus RTU VHDL implementation. This is the portion of the algorithm that processes and assesses Modbus RTU packets sent from the master device and received by the slave device instantiation. The remainder of this subsection will continuously reference the state diagram of the command request processing NSL algorithm below in Figure 3.38.

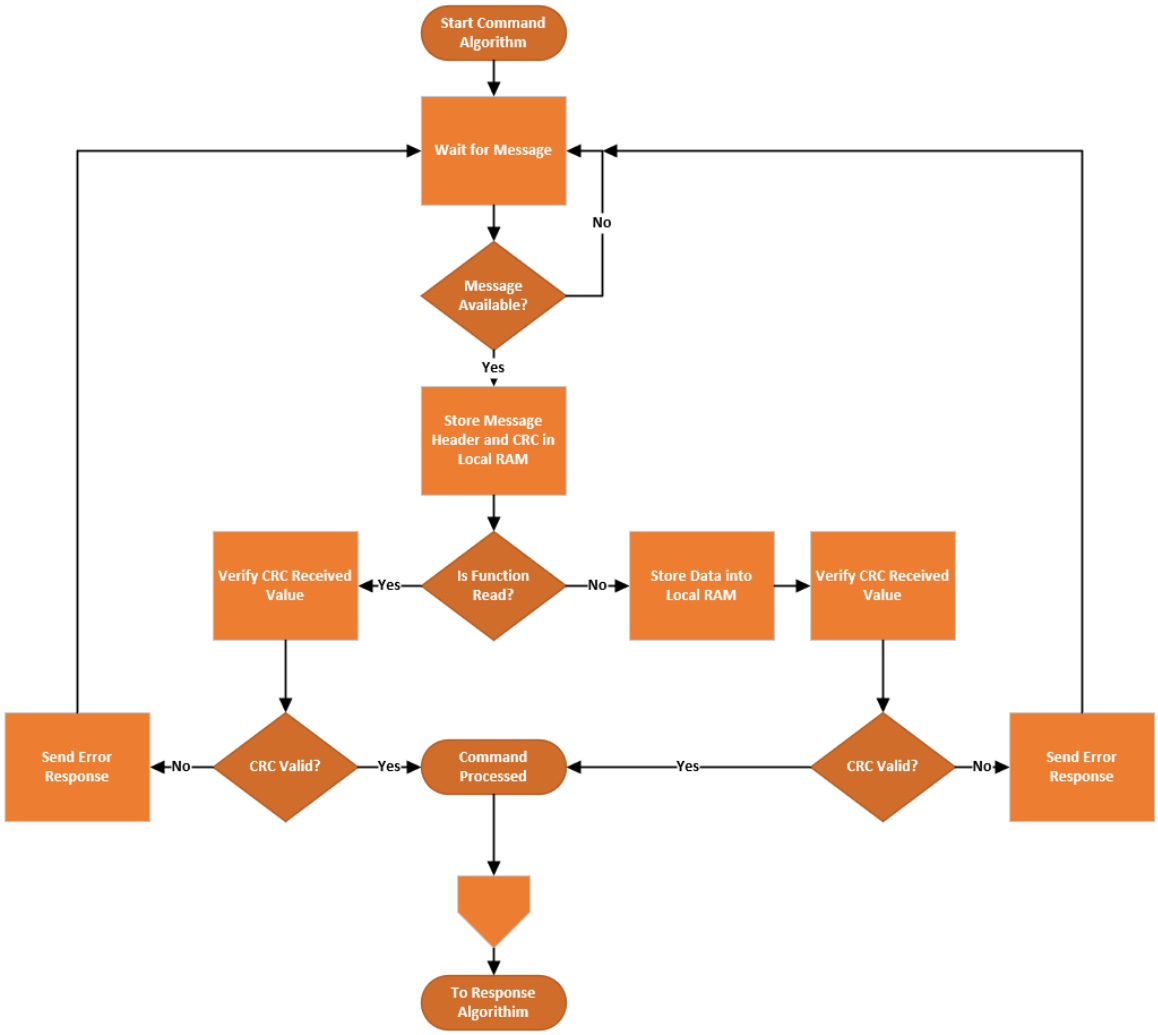


Figure 3.38 - State diagram of the next-state-logic algorithm used for the processing of a Modbus RTU command request received by the slave device

Receiving Message

This portion of the NSL algorithm begins by waiting in a cyclical state for a signal to be asserted from the bus interface to indicate that a message is available in the incoming FIFO register. When an incoming Modbus RTU message is sent from the master device, the signals that carry the packet contents first travel across the common bus line. The message transmitted to the slave device has its packet contents stored in the incoming FIFO register of the Modbus RTU instantiation. Once the incoming FIFO register has message contents available, the bus interface asserts a signal indicating there are available message contents ready to be processed.

Once a message is available in the incoming FIFO register and the signal indicating available message contents is asserted, a process to store the incoming message components in the local memory begins. First, the message header contents of the device address, function code, Hi and Lo byte of the address to index, and the Hi and Lo byte of the number of registers is stored in the local memory. Next, the received CRC value is stored in the local memory. The NSL algorithm then branches into two potential paths depending on whether the command request is a write or read request. This is determined by evaluating the value of the function code stored in the local memory, and the algorithm routes the process to the corresponding case statement pathway. An excerpt of the VHDL code that routes the NSL algorithm based on the contents in the local memory indicating a read or write request can be seen below in Figure 3.39.

```
918             when S51=>
919                 if(RAM_Data_Out = X"03") then      --Check Cmd (Read)
920                     Is_Read <= X"01";
921                     LD_Is_Read <= '1';
922                     NS_FIFO_Bus <= S60;
923                 elsif(RAM_Data_Out = X"10") then   --Check Cmd (Write)
924                     Is_Read <= X"00";
925                     LD_Is_Read <= '1';
926                     NS_FIFO_Bus <= S52;
927                 else                               --Check Cmd (Invalid Data)
928                     NS_FIFO_Bus <= S0;
929                 end if;
```

Figure 3.39 – VHDL code in NSL algorithm that routes state pathway based on read or write request

Read Request

If the request received from the master device is a read request, then the NSL algorithm proceeds by verifying the contents and integrity of the message. This is accomplished by recomputing the CRC calculation based on the received message. If the calculated CRC value matches the received CRC value, then the message is considered valid, and the NSL algorithm proceeds to the request fulfillment portion of the algorithm covered in subsection 3.3.4. If the calculated CRC value does not match the received CRC value, the NSL algorithm routes to an algorithm that forms an error response and sends the error message back to the master device to indicate the received message is invalid.

Write Request

If the request received from the master device is a write request, then the NSL algorithm stores the received data from the incoming FIFO register in the local memory. Once the data contents are stored in the local memory, the NSL algorithm proceeds by verifying the contents and integrity of the message. This is accomplished by recomputing the CRC calculation based on the received message. If the calculated CRC value matches the received CRC value, then the message is considered valid, and the NSL algorithm proceeds to the request fulfillment portion of the algorithm covered in subsection 3.3.4. If the calculated CRC value does not match the received CRC value, the NSL algorithm routes to an algorithm that forms an error response and sends the error message back to the master device to indicate the message was invalid.

3.3.4 – Command Response Processing Algorithm

This subsection will discuss the portion of the NSL algorithm that processes the command responses within the Modbus RTU VHDL implementation. This is the portion of the algorithm that fulfills the command request and forms the Modbus RTU response packet that is sent from the slave device to the master device. The remainder of this subsection will continuously reference the state diagram of the command request processing NSL algorithm below in Figure 3.40.

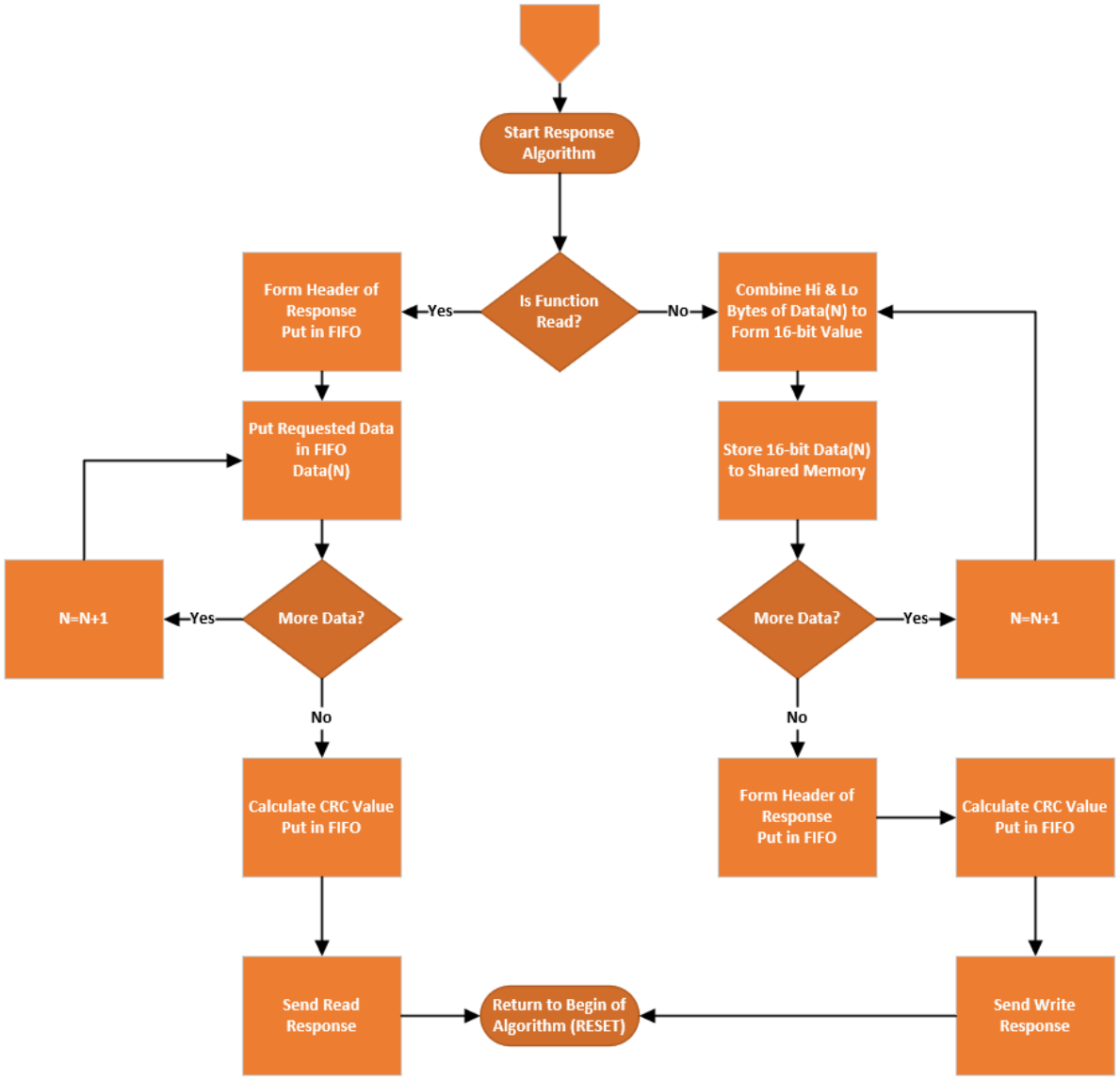


Figure 3.40 - State diagram of the next-state-logic algorithm used for the processing and fulfillment of the Modbus RTU command response received by the slave device

Sending Response

This portion of the NSL algorithm begins by separating the command request fulfillment and response process based on whether the request was a read or write command. The basic pathway of the read and write command fulfillment consists of preparing the data to be read out or written in, forming the corresponding message header, and calculating the CRC value to be attached to the packet. All these

message components are put in the outgoing FIFO register, and the bus interface sends the constructed Modbus RTU packet along the shared common serial bus line to the correct master device. The remainder of this subsection will discuss in detail both of the NSL algorithm pathways based on the read or write fulfillment and response processes.

Read Response

If the command request to be fulfilled is a read request, the Modbus RTU VHDL implementation must provide the requested data to the master device. The first steps in the NSL algorithm for this process are to form the header contents of the read response and place them in the outgoing FIFO register. The first contents of the header consist of repeating the original device address and function code. The final component of the Modbus RTU packet header for the read response is to include the value of how many bytes of data will be included in the Modbus RTU packet. This is an 8-bit value that indicates to the master device how much data it needs to expect during the transmission of this packet.

Once the header components of the packet are placed in the outgoing FIFO register, the data to be read to the requesting master device needs to be prepared. To retrieve this data, the NSL algorithm uses the received value of the starting address to index and the number of requested registers stored in the local RAM during the previous portion of the algorithm. These values are then used to retrieve the data from the shared memory. The retrieving of data is carried out by the bus interface and made available to the NSL algorithm. Next, the specified data values are populated into the outgoing FIFO register and the local RAM. Once all requested data has been retrieved from the shared memory, the CRC calculation is performed, and the resulting value is placed into the outgoing FIFO register. As this is the last component of the Modbus RTU packet that needs to be created, a signal is sent to the bus interface to indicate the message is available and ready to send. Finally, the bus interface sends the formed Modbus RTU response message containing the requested data to the master device.

Write Response

If the command request to be fulfilled is a write request, the Modbus RTU VHDL implementation must store the data in the shared memory at the specified register addresses and respond to the requesting master device that the action was successfully carried out. Before the data can be written into the shared memory of the FPGA, it must first be converted into 16-bit values. The incoming data values of the Modbus RTU write packet are received in 8-bit values with Hi and Lo byte components. The NSL algorithm handles this by first combining the Lo and Hi bytes into a 16-bit value and then subsequently storing this 16-bit value in the shared memory of the FPGA. This cyclical process is carried out for all received data values as the received data is processed.

Once the received data has been written to the shared memory of the FPGA, the NSL algorithm proceeds by forming the header components of the Modbus RTU response packet. The header components consist of repeating the original device address, function code, and Hi and Lo byte of the address of the first register to index. Next, a 16-bit value of the number of recorded registers will be included in the response packet. This value is calculated within the NSL algorithm using an incremental counter in conjunction with the cyclical process of storing data in the shared memory of the FPGA. This value is included so the master device can verify that the correct number of registers was recorded. This 16-bit value is broken into Hi and Lo bytes and placed into the outgoing FIFO register.

The final component to creating the response message is computing and including the CRC value in the packet. The CRC is calculated based on the message packet components created and placed into the outgoing FIFO register. As this is the last component of the Modbus RTU packet that needs to be created, a signal is sent to the bus interface to indicate the message is available and ready to send. Finally, the bus interface sends the formed Modbus RTU response message containing the requested data to the master device.

3.3.5 – Cyclic Redundancy Check

The NSL algorithm portion that carries out the calculations for the CRC-16 value is relatively complex to implement in VHDL due to the various content structures of a Modbus RTU packet. Within this Modbus RTU implementation, the CRC-16 value must be calculated to verify message integrity for

incoming read and write messages, read and write responses, and error-handling responses. To address this challenge, three different implementations for the CRC-16 calculations were made using NSL. As the CRC-16 algorithm utilizes nested for loops, it became more effective to write the HDL code for each of these three cases individually instead of creating additional nested layers. A unique CRC-16 algorithm was created for each of the previously mentioned scenarios.

3.3.6 – Transmission Error Handling

If a Modbus RTU packet is received from the master device and the CRC value calculated within the NSL algorithm does not match the CRC value received in the incoming message, the message must be declared invalid. If a message is declared invalid, the NSL algorithm will discard the message and will not fulfill the received request. To communicate back to the master device that the message was invalid, an error response Modbus RTU message is formed and sent.

The contents of the error response consist of five bytes of data. First, the originally received device address is repeated and placed into the outgoing FIFO register. Next, the original value of the function code byte has its highest bit incremented by ‘1’ and is placed in the outgoing FIFO register. For example, if the original function code is “0x03,” representing a read request, the binary value is “0b00000011”. The highest bit is incremented, resulting in a binary value of “0b10000011” or hexadecimal “0x83”. This newly modified value in the response packet indicates to the master device that an error has occurred. The next part of the response packet is an 8-bit value containing the error code placed into the outgoing FIFO register. For this work, the error code did not need to be unique to the error experienced, and a value of “0x01” was utilized. This helps reduce the amount of code required in this Modbus RTU implementation while also allowing the user to receive an indication of an error, and further investigation may be necessary. The final component of the error response packet is the calculated CRC value. This value is calculated and placed into the outgoing FIFO register. As this is the last component of the error response Modbus RTU packet that needs to be created, a signal is sent to the bus interface to indicate the message is available and ready to send. Finally, the bus interface sends the formed Modbus RTU response message containing the requested data to the master device.

3.4 – Modbus RTU, Multiple Instantiations

The custom Modbus RTU implementation of this work operates on the RS-232 standard and is a single point-of-service system. This means the Modbus RTU communication protocol within the FPGA is designed to only communicate with a single master device and only fulfill the requests from that device. To utilize multiple communication pathways with the Modbus RTU communication protocol, the code to create this Modbus RTU implementation has been instantiated multiple times. There are three instantiations of this embedded Modbus RTU module in the SETO project system. A network diagram highlighting the location and connection of the three embedded Modbus RTU instantiations can be seen below in Figure 3.41. This figure illustrates the SETO module network interconnections and highlights the three embedded Modbus RTU modules of the system. These three module instantiations are designed to serve only the requests from their specific master devices. Although the common serial bus is shared between instantiations, the requested commands will be processed accordingly.

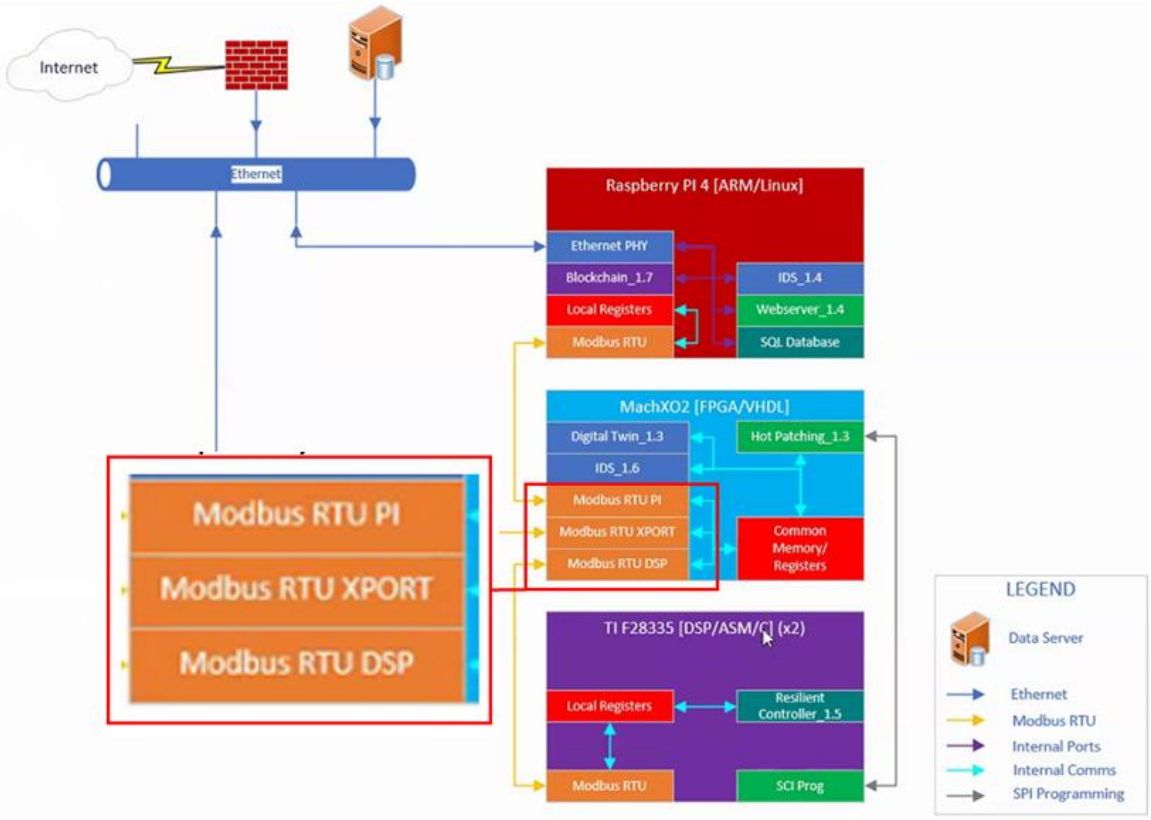


Figure 3.41 - SETO module network interconnection diagram highlighting the three embedded Modbus RTU instantiations servicing the Raspberry Pi, Ethernet connection, and DSPs

The communication pathways for these three modules are supported through various infrastructure components. The first connection for the Raspberry Pi is serviced through a mini-USB connection via the TI port on the UCB, seen below in Figure 3.42. The second connection to the local network through the Ethernet physical layer is supported through an XPORT module on the UCB. This XPORT module converts the Modbus RTU packets into Modbus TCP packets and vice versa. This communication is then utilized with various systems on the network, such as the web server, data logger, and LabVIEW HMI used by the operator. The third connection for the two DSPs is an internal connection made within the UCB. Various input and output pins within the FPGA are connected to the DSPs via wire tracing in the printed circuit board (PCB).

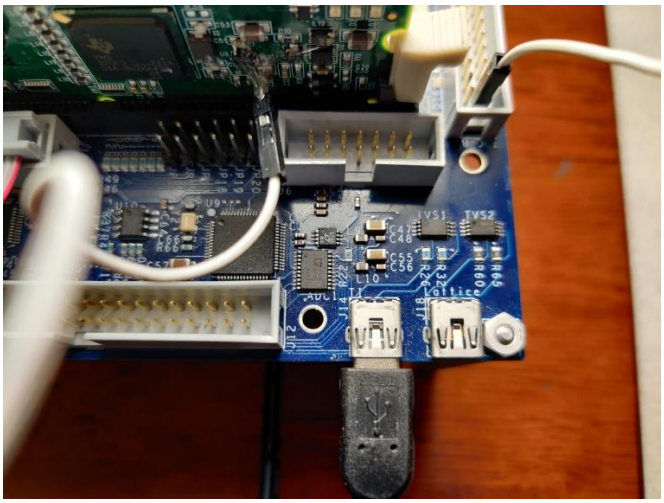


Figure 3.42 – Mini-USB connection for Modbus RTU communication with Raspberry Pi through TI port on UCB

Chapter 4

Test Setup and Experimental Results

4.1 – Introduction

The work presented in this research addresses the challenge of creating an embedded Modbus RTU communication protocol architecture with respect to the bounds of finite available resources. Additionally, it is valuable to be aware of the time investments required for researchers to collaborate and integrate their respective works. The proposed design addresses these challenges by providing a standardized communication pathway common to industrial control systems, such as solar photovoltaic farms, while requiring fewer resources than typical embedded system implementations.

This section will provide the results of the various experimental setups used within the testing and development process of this embedded Modbus RTU implementation. As the design process was relatively complex, there were multiple testing and troubleshooting milestones before the final product. This testing and troubleshooting process involved five unique testing phases. During the design and development, the five testing phases were first VHDL simulations, then testing within a developmental board equipped with an FPGA, next using the UCB of the SETO project, followed by resiliency testing with the Python programming language, and finally testing with the bootloader and hotpatching LabVIEW HMI. A relative timeline of these testing phases with their associated chapters can be seen below in Figure 4.43. Most of this chapter will discuss the various testing setups and incremental results of those tests. The final subsection of this chapter will discuss the results with respect to the resource-efficient implementation of this embedded communication protocol architecture.

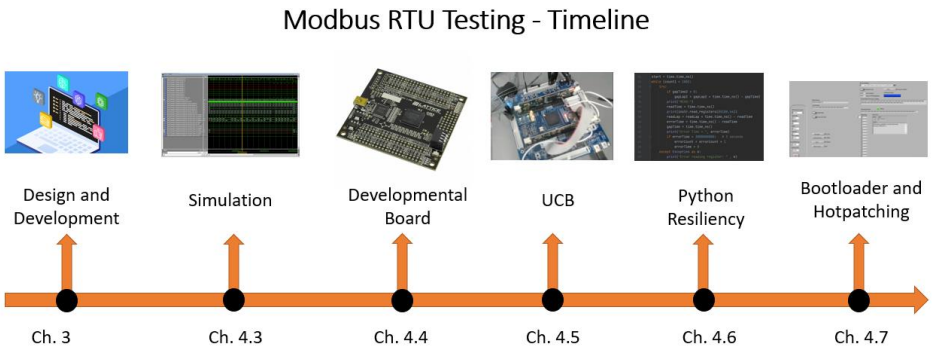


Figure 4.43 - Timeline of the testing process for the embedded Modbus RTU implementation

4.2 – Test Setup

The Modbus RTU communication protocol and the associated hardware this system architecture operates on are described in Chapter 2. As these systems are individually relatively complex, integrating these various elements requires a multitude of software and hardware to perform adequate testing. This subsection will discuss the various hardware and software-based components used throughout the various testing setups. A detailed discussion of each testing setup and its use of these components will be in the corresponding subsections discussing each testing phase.

Hardware

The hardware used in this testing setup consists of the UCB, a Lattice breakout board equipped with a MachX03D FPGA, a Raspberry Pi 3 operating on Linux, and a personal computer operating on Windows. The UCB is the device that holds the final version of the embedded Modbus RTU communication protocol and acts as the central routing fabric and controller for the SETO project. The Lattice breakout board is a device that supplements the design process of FPGA-based projects. This breakout board contains an FPGA and various pinouts that can be utilized. This device aimed to test early versions of the embedded communication protocol. While the FPGA chip in this device is not the same as the model of the UCB, the functionality for this testing iteration was equivalent to the FPGA within the UCB.

The two types of computers used to perform the various tests of this work are a Raspberry Pi 3 and a personal computer. These devices operate on the Linux and Windows operating systems, respectively. Testing on these two devices allowed for different testing environments to ensure the embedded communication protocol implementation was functional on the platforms utilized by different modules of the SETO project. Additionally, the method by which each operating system manages its serial bus interfaces differs. This difference will be discussed more in Chapter 4.6.

Software

This testing setup uses the Lattice Diamond software, various created LabVIEW HMIs, a serial communication testing application called XCTU, and the “MinimalModbus” Python package. The Lattice Diamond software is used throughout the development and testing phases. This software is the workspace and environment used to code within VHDL and program the coding into the FPGA. This software also supports the simulation application ModelSim, which is used to perform simulations of the embedded Modbus RTU implementation during the development and testing phases. The various LabVIEW HMIs were created for the UCB testing and bootloader and hotpatching testing phases by providing an interface to interact with the embedded Modbus RTU architecture. These HMIs are created to be capable of communicating through the Modbus RTU protocol. The XCTU application is a basic serial communication troubleshooting tool that allows for testing the embedded Modbus RTU functionality. The “MinimalModbus” Python package allows for interfacing with Modbus RTU devices over serial ports, such as USB, using the Python programming language.

4.3 – Simulation Testing and Results

The simulation testing phase using the ModelSim application within the Lattice Diamond software was the first phase of testing and troubleshooting during the development process of this work. The main purpose this phase of testing served was to troubleshoot bugs and gain informative perspectives while developing the embedded Modbus RTU architecture. These simulations through ModelSim allow for the fine-grain perspective of signals and variables. This zoomed-in view provided through the simulation process allows for viewing how these signals and variables change on a significantly smaller scale relative to the 25 MHz clock period of 40 nanoseconds. This high-fidelity simulation gives insight into how each variable and signal changes within the design and at what point they change.

The nature of designing embedded systems within FPGAs creates real physical logical and hardware-based connections within the device’s configurable logic blocks. As the laws of physics don’t allow for the instantaneous transition of an electrical voltage [35], the timing for signal propagation is an imperative limitation of the laws of physics to be aware of. Additionally, metastability [36] is of potential concern if these delayed signal propagations leave a register or device in a non-steady state at the time

of reading or using these signals. As VHDL is a hardware description language, the code describes the architecture of physical and logical components. In some cases, consideration for data propagation delay needs to be taken. The simulation environment provided by the ModelSim software helps facilitate a perspective aware of these considerations. An example of a case where the register data of a variable is not available within the typical one-clock cycle is when storing data in RAM. Through trial and error, it was found that this process takes two clock cycles for the data to be available on the logical hardware to be stored in RAM. To overcome this, two nearly identical and sequential case statements are created, as seen in Figure 4.44 below. The ModelSim simulation waveform of this coding can be seen below in Figure 4.45.

```
1434        when S29=>
1435            RAM_address <= Rcv_Cnt_Out + Rcv_Cnt_Out + 8;
1436            Temp_Data_Low <= RAM_Data_Out;
1437            --LD_Temp_Data_Low <='1';
1438            NS_FIFO_Bus <= S30;
1439
1440        when S30=>
1441            RAM_address <= Rcv_Cnt_Out + Rcv_Cnt_Out + 8;
1442            Temp_Data_Low <= RAM_Data_Out;
1443            LD_Temp_Data_Low <='1';
1444            NS_FIFO_Bus <= S31;
```

Figure 4.44 – VHDL code of nearly identical case statements to account for propagation delay when storing values in RAM

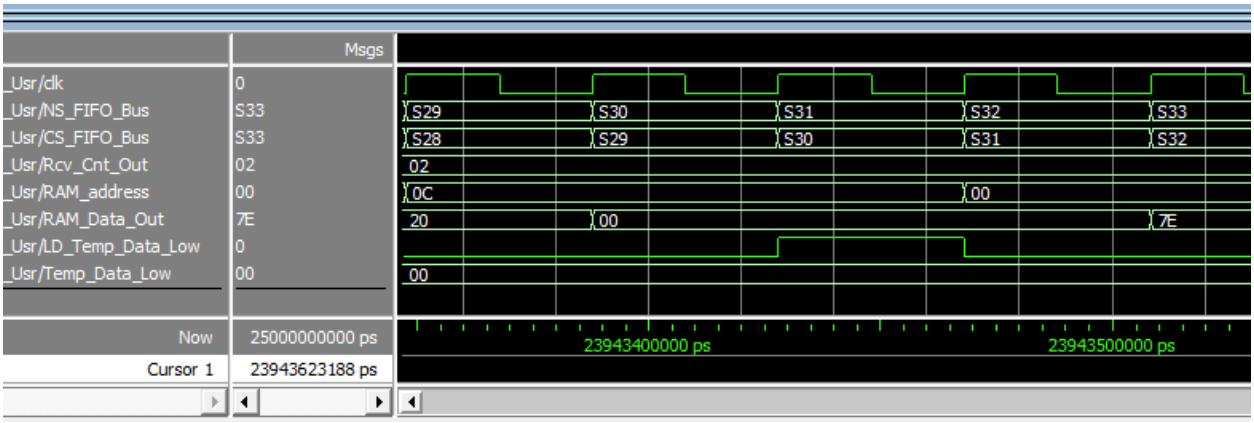


Figure 4.45 – VHDL simulation of code from Figure 4.44, depicting the gate-level simulation of redundant case statements to allow RAM data to be available on physical hardware before storing

The high-fidelity simulation waveforms, seen below in Figure 4.46, allow for viewing case statements and register values with a high amount of detail. The figure below depicts five values over approximately seven clock cycles in the VHDL testbench used to verify the CRC functionality. The process of the NSL algorithm in the figure is the recalculation and the verification of the received CRC value. These five signals are the clock signal (clk), the current state of the NSL algorithm (CS_FIFO_Bus), the value holding the received CRC value (Chk_Sum_reg_o), the register holding the temporary value of the ongoing CRC calculation (CRC16_Temp_reg_o), and a register value used to indicate if the CRC calculation matches the received CRC value (CRC_Pass_reg_o). The current state of the NSL algorithm switches to the subsequent case statement on the low-to-high transition of the clock signal.

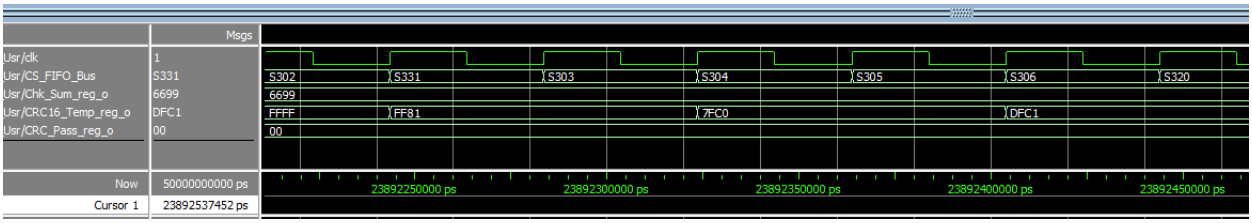


Figure 4.46 – High fidelity simulation results of CRC calculations, showcasing case statement, and CRC register value transitions

Storing Data into Local RAM

One of the most pertinent resources the NSL algorithm uses to process commands, fulfill requests, and formulate the appropriate responses is the ability to store data in a local RAM or memory. Access to memory unique to each instantiation of the Modbus RTU implementation gives that unique module a location to store data unique to that system without adding the complexities of utilizing a shared memory for these backend operations. The insights given by the simulation testing aided the design and development of this work by adding perspectives on where and how the data stored in this local memory is available.

As previously mentioned in Chapter 3, as an incoming Modbus RTU message enters the communication module via the incoming FIFO register, it is subsequently stored in the local memory. The contents of the FIFO are popped out byte-by-byte and sequentially placed into the module's local memory, as seen in the simulation screenshot of Figure 4.47 below. The first component from the incoming FIFO register is the 8-bit value of the device address placed in the local RAM at address '0' and can be seen in the hexadecimal radix of the figure. Each address of the local RAM corresponds to a particular byte in the structure of the Modbus RTU packet. As the structure of the Modbus RTU packets varies between read or write and requests or commands, not every address location is used by every packet. Within this register mapping, each component has a static register address except for the data and CRC bytes. As the total length of the data is a dynamic value, the addresses of the register mapping used to store this data are also dynamic. Additionally, the CRC values are referenced within the register mapping according to the number of registers within the data contents. The CRC's reference address is based on the total number of data registers, N, where the address values of the Hi and Lo-byte of the CRC variables are N+7 and N+8, respectively. Table 3 below depicts the register mapping used for the local RAM when storing and retrieving data during the processing and response processes.

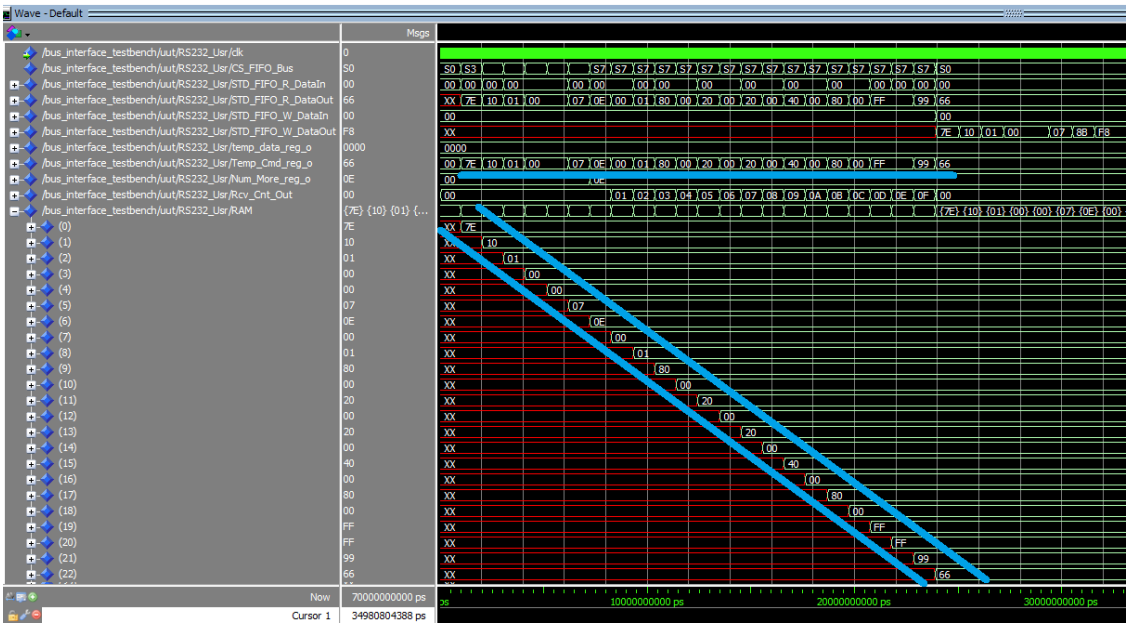


Figure 4.47 – Simulation results showcasing the NSL algorithm storing the incoming message components (top horizontal blue line) into the local RAM (diagonal blue lines) to be used for various references and calculations

Table 3 – Register mapping for local RAM of Modbus RTU packet structure components

Address in RAM	Variable Stored	Address in RAM	Variable Stored
00	Device Address	**N = Increment Number of Total Data**	
01	Function Code	**07**	Data Hi-Byte
02	Starting Address Hi	**08**	Data Lo-Byte
03	Starting Address Lo	**07+N**	Data + N Hi-Byte
04	Number of Registers Hi	**08+N**	Data + N Lo-Byte
05	Number of Registers Lo	**07+N, Final**	CRC Hi-Byte
06	Number of Byte More	**08+N, Final**	CRC Lo-Byte

CRC Calculations and Verifications

The CRC calculation and verification process is one of the vital components of the Modbus RTU communication protocol that makes it well known to be a reliable and resilient method of communication within industrial control systems. Creating the three CRC verification or creation algorithms previously mentioned in Chapter 3 was one of the more difficult components of the Modbus RTU communication protocol to implement. It was pertinent to minimize the number of nested loops within the NSL algorithm when verifying or calculating a CRC value. The simulations of this portion of the NSL algorithm offered valuable insights into how the CRC process behaved, which helped design the corresponding case statements within VHDL.

The utilization of the simulation software's advanced feature to display the FPGA's clock-by-clock cycles can provide an unparalleled advantage in terms of gaining access to intricate and precise details of the state machine’s values and variables. This is because it enables the observer to view the minute fluctuations in the FPGA's functions, which would have been difficult to notice otherwise. By taking advantage of this feature, one can acquire a much deeper insight into the working of the state machine and make informed decisions based on this data to assist with the design and troubleshooting process. This ability to observe the FPGA's state at the scale of clock-by-clock cycles offers a significant boost to the precision and accuracy of the system's analysis and testing processes. This high-fidelity waveform

perspective offered by the simulation software of the CRC validation algorithm completing its verification process can be seen below in Figure 4.48. In this waveform, the “Chk_Sum_reg_o” variable highlighted in red contains the received 16-bit CRC value from an incoming message. This value is recalculated through the CRC NSL algorithm using the received contents of the Modbus RTU message and is stored in the “CRC16_Temp_reg_o” variable highlighted in blue. If the value of these two variables matches, the message’s contents have been verified, and a bit is asserted in the “CRC_Pass_reg_o” variable, highlighted in white, to indicate that the message contents have been confirmed. The final confirmation of the CRC calculation for response messages and validation for received messages can be seen in Figure 4.49 and Figure 4.50, respectively.

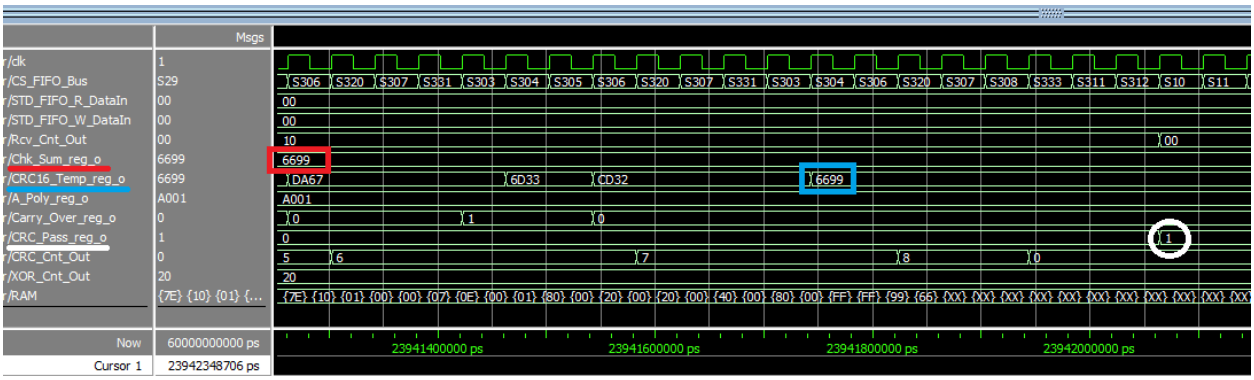


Figure 4.48 – Simulation waveform of register-level CRC verification process when validating a received CRC from an incoming message, the received value (red) is recalculated (blue) based on the received message contents, and a flag (white) is asserted if the values match

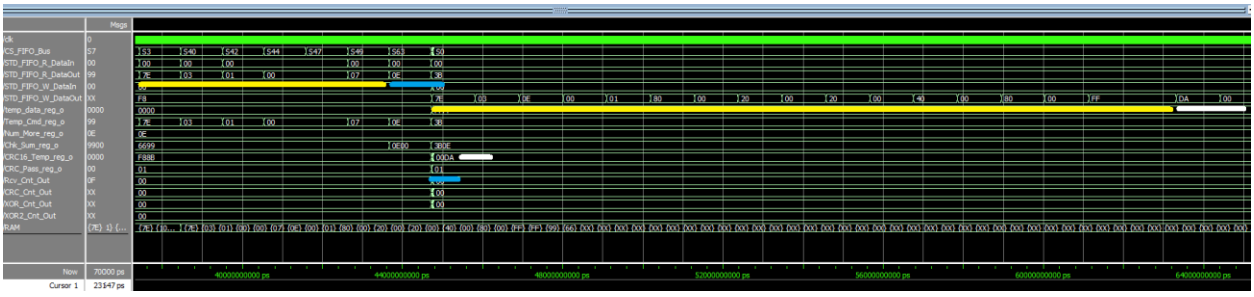


Figure 4.49 – Simulation of a read command depicting the results of the verification of the received CRC value (blue) and the calculation of the response CRC value (white), the yellow bars represent the data that is used within the CRC calculation for the subsequent highlighted CRC value

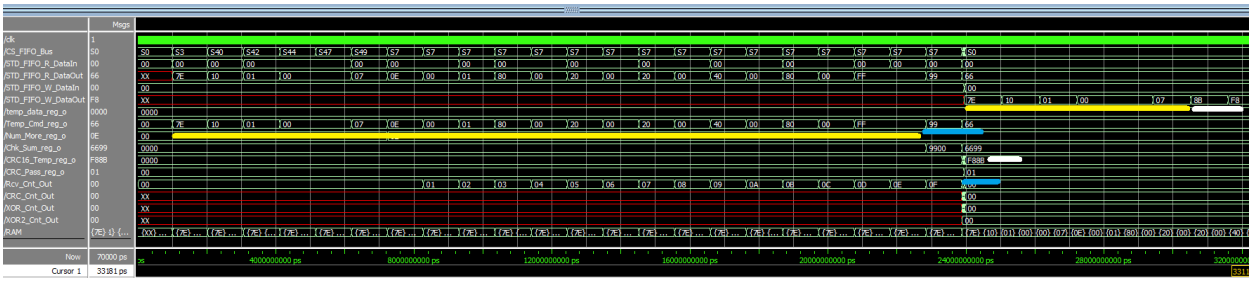


Figure 4.50 - Simulation of a write command depicting the results of the verification of the received CRC value (blue) and the calculation of the response CRC value (white), the yellow bars represent the data that is used within the CRC calculation for the subsequent highlighted CRC value

Final Simulation Test – FPGA, DSP, and RPi Instantiations

The final portion of the simulation testing requires instantiating the developed embedded Modbus RTU implementation into three separate modular communication pathways. As mentioned in Chapter 3, the three systems within the SETO project that require communication with the FPGA's routing fabric are the XPORT for HMI use, the internal wire connections to the DSPs, and the serial port for the Raspberry Pi. These three modules are instantiated using the inherent capability of VHDL to create a copy of a design entity within an embedded architecture covered in Chapter 2.

To incorporate three different modules of the embedded Modbus RTU implementation, the signal pinouts needed to be coordinated and assigned. Once the three sets of transmission and receiving pins are assigned, the VHDL testbench is created to validate the functionality in a simulated environment of the three embedded Modbus RTU instantiations. Below in Figure 4.51 are the simulation waveforms depicting the operation and response of the three modules to write commands sent through each communication pathway. The function commands and responses for each module can be seen with the XPORT in white, the DSP in blue, and the RPi in yellow.

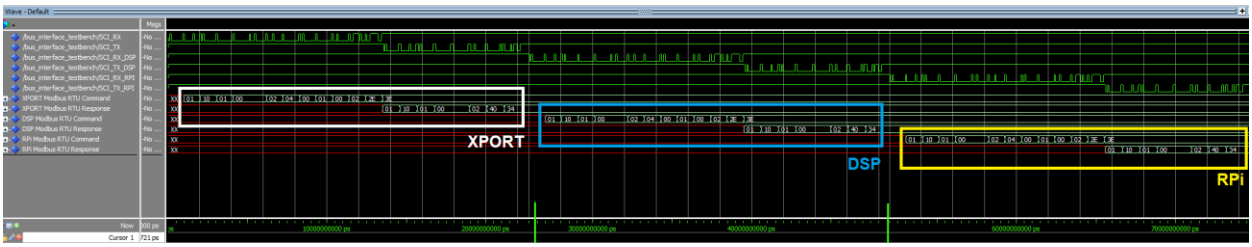


Figure 4.51 – Simulation waveform of master device write commands and slave device responses for three Modbus RTU module instantiations: XPORT module (white), DSP module (blue), and Raspberry Pi module (yellow)

4.4 – Development Board Testing and Results

The developmental testing phase using the Lattice MachX03D Breakout Board, also referred to as a development board and seen below in Figure 4.52, was the initial phase of testing and troubleshooting the NSL algorithm on physical hardware during the coding development process. The main purpose this phase of testing served was for the preliminary confirmation of the NSL algorithm working on hardware and to identify any potential timing issues while developing the embedded Modbus RTU architecture. As the environment the simulation testing phase provides is ideal, it is imperative to incorporate real hardware testing into the design and development process. This subsection will discuss some of the processes, challenges, and results of this testing phase.

Figure 4.52 – Lattice breakout board equipped with a MachX03D FPGA (model LCMX03D-9400HC)

As mentioned in the previous subsection, the issue of metastability and data validity can present itself in a physical system. This phenomenon's first occurrence emerged while testing basic Modbus RTU write commands. The response message received from the embedded Modbus RTU module contained an invalid value for the CRC command. Upon investigation, it was found that the referenced RAM address of the local memory was occasionally incorrect, which resulted in an incorrect CRC calculation. The invalid data was most likely a result of the physical hardware not having access to the correct signal value when the NSL algorithm referenced the value. This issue was remedied by utilizing redundant case statements in the NSL algorithm like the redundant statements previously referred to in Figure 4.44.

The second challenge during this initial hardware testing phase was the timing of message transmissions. As mentioned in Chapter 2, the Modbus RTU protocol follows a message timing and framing structure. The timing issue experienced was an inconsistency in received message responses. The algorithm would occasionally enter a fringe state when receiving back-to-back incoming messages and remain stuck until additional data was fed into the incoming FIFO register. The NSL algorithm could execute an error response. To properly address this challenge, an additional 3.5-character timing delay feature was added to the NSL algorithm code at the end of the response portion of the NSL algorithm. This timing delay addition appeared to assist with the reset state process of the NSL algorithm.

Once the data metastability and timing issues were addressed, the Modbus RTU algorithm appeared functional in the initial hardware testing phase. The operation of a write command and the subsequent reading of that stored data through a read command can be seen below in Figure 4.53. In this figure, the blue text is the command manually sent using the XCTU software, and the red text is the response received via USB from the embedded Modbus RTU algorithm. The first manual command writes the data in 16-bit registers of “0x0001, 0x8000, 0x2000, 0x2000, 0x4000, 0x08000, 0xFFFF” to seven registers in the shared memory starting at address 0x0100. The subsequent manual command requests the data that was written to the shared memory to be read back through XCTU. To rule out the possibility of the algorithm only parroting received data, the NSL algorithm was further tested with consecutive write and read commands storing and accessing data from dynamic and overlapping registers. The test results below in Figure 4.54 confirmed that the NSL algorithm passed this more

scrupulous testing. This final test concluded the initial testing phase of the NSL algorithm on the hardware of the developmental board.

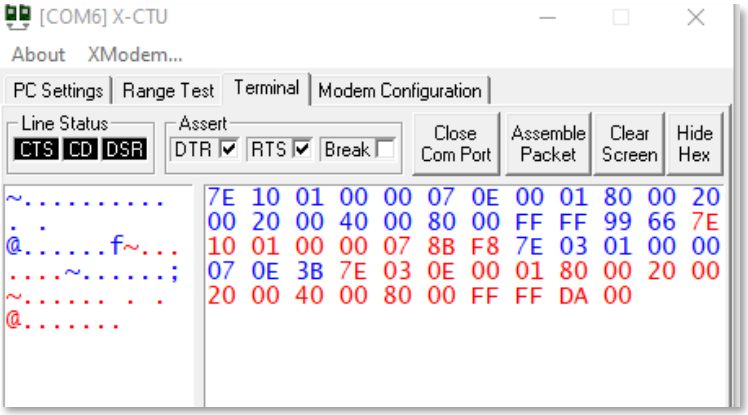


Figure 4.53 – Communication between the XCTU software and the embedded Modbus RTU module via a USB connection, the blue bytes in hexadecimal represent the manually input command request while the red bytes are the response from the FPGA

Figure 4.54 – Additional image of communication between the XCTU software and the embedded Modbus RTU; each manual command (blue) is requesting to read a different number of registers while the responses (red) deliver accordingly

4.5 – UCB Testing and Results

The UCB testing phase is the second assessment of the NSL algorithm utilizing physical hardware and the infrastructure of the final physical hardware setup for the SETO project. This section will discuss the testing of the NSL algorithm in a setup that utilizes all three instantiations of the embedded Modbus RTU during a single test. A diagram of the integration scheme used for these three instantiations can be seen below in Figure 4.55. This testing configuration allows for simultaneous

reading and writing from multiple devices. Additionally, this testing phase confirms the accuracy and availability of data being stored in the shared memory of the FPGA. The communication pathways of the three instantiations of the embedded Modbus RTU implementation consist of a USB connection to a Linux device equipped with Python, a USB connection via an external FTDI chip to the XCTU software, and the XPORT ethernet connection to a LabVIEW HMI.

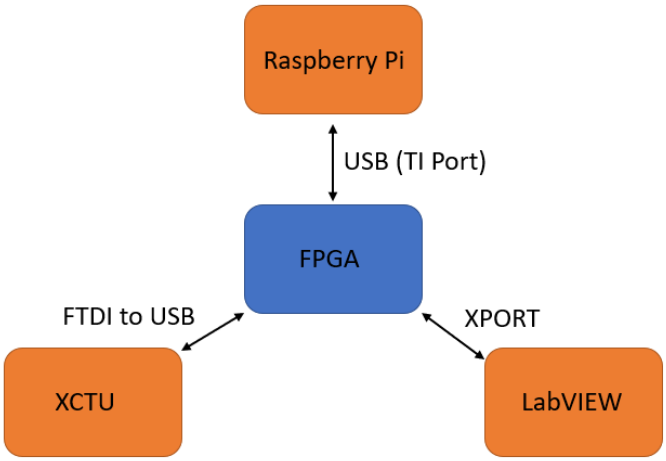


Figure 4.55 – Connection diagram of three embedded Modbus RTU instantiations for the UCB testing in the hardware phase

Serial Port Instantiation (RPi)

The instantiation that operates through the TI serial port connection via USB connects to the Raspberry Pi device. For this testing phase, the Raspberry Pi was running on the Kali Linux operating system. The RPi uses the “MinimalModbus” Python package to communicate with this Modbus RTU instantiation. A basic exert of this “MinimalModbus” code can be seen in Figure 4.56, which performs a read register command followed by a write register command. The “MinimalModbus” Python code acts as the master device for the embedded slave device. The Python code created for this Modbus RTU instantiation was set on a delay-controlled loop to read, modify, and store the data back into the FPGA. This was done by reading sixteen 16-bit registers from the shared memory of the FPGA and storing these values into a local Python list variable. These values were then incremented by one, and a write command was made to store these modified values back into the shared memory of the FPGA.

```
RegList = [0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15]


try:
    print("READ:")
    print(instr.read_registers(0X00,8))

except Exception as e:
    print("Error reading register: ", e)

print("\nSleeping for 2 seconds\n")
time.sleep(2)

try:
    print("WRITE")
    instr.write_registers(0X00, RegList)

except Exception as e:
    print("Error writing Coil: ", e)

```

Figure 4.56 – Basic Python code using the "MinimalModbus" package to use Modbus RTU to read and write with registers within the FPGA through embedded Modbus RTU

External FTDI Instantiation (XCTU)

The instantiation that operates through the external FTDI chip, seen in Figure 4.57 and Figure 4.58, connects via USB to a personal computer running the XCTU software. The XCTU software was used to manually read variables to confirm that the values stored by the other instantiations were available and correct. Additionally, XCTU was used to manually store values in various registers of the shared memory of the FPGA.

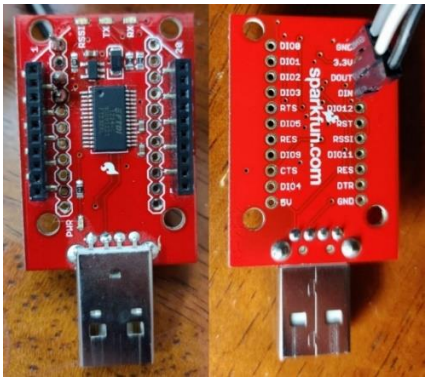


Figure 4.57 – External FTDI chip for converting between serial communication and the USB standard

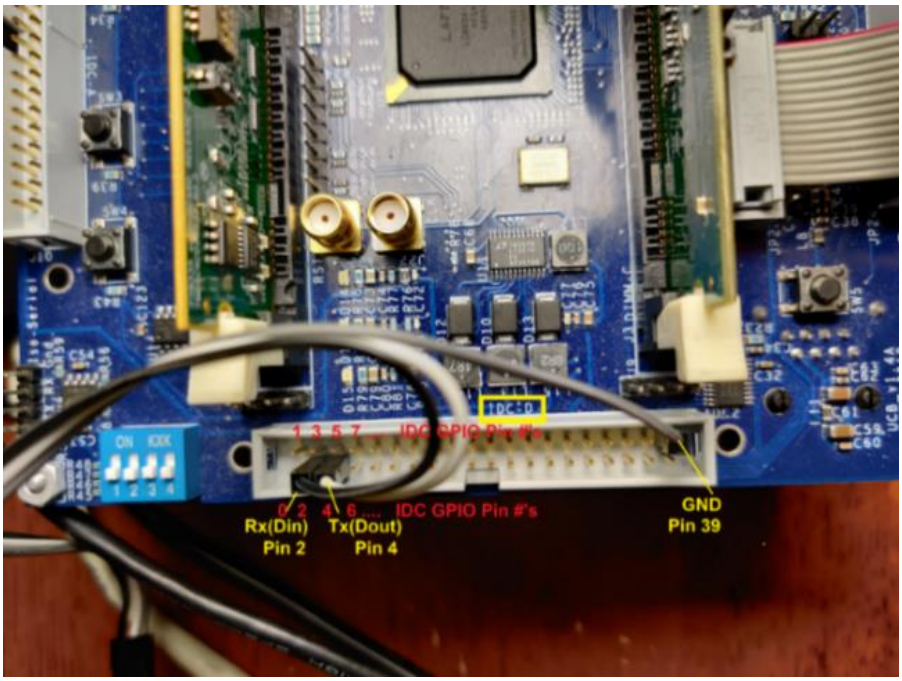


Figure 4.58 – Connection of pins from IDC-D on UCB to jumper wires of FTDI chip (seen in the top right of Figure 4.57) utilizes the signals of Tx, Rx, and a ground reference

XPORT Instantiation (LabVIEW)

The instantiation that operates through the XPORT module connects to a personal computer via an ethernet connection. The personal computer runs the LabVIEW software, where an HMI was created to connect to the embedded Modbus RTU components. This LabVIEW HMI allows reading and writing registers in a loop. This is used to confirm the accuracy and availability of data stored in the shared memory of the FPGA in conjunction with the other instantiations. Additionally, the register address to read and write data to and from can be changed dynamically to confirm data integrity and availability.

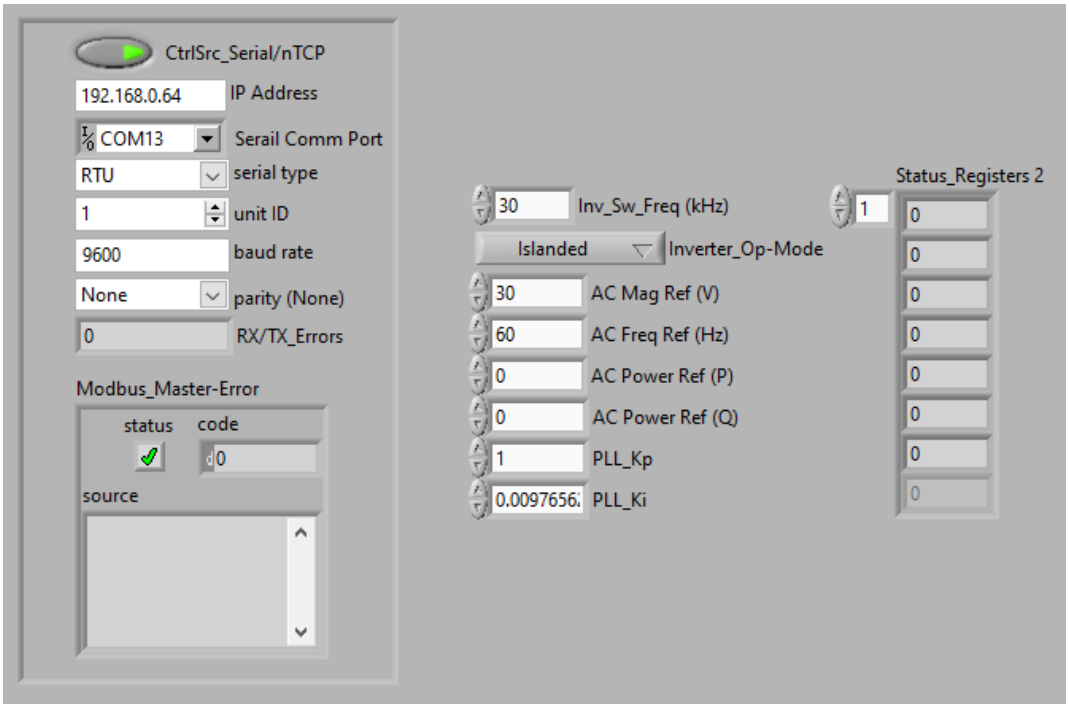


Figure 4.59 – LabVIEW interface created to facilitate the testing of the embedded Modbus RTU algorithm, three columns (from left to right) represent the configuration setup and error reporting, the register values to write, and the register values being read, respectively

Results of Three Modbus RTU Instantiation Integration

To validate the accuracy and availability of the data, the three embedded Modbus RTU instantiations were set up to allow for the dynamic changing of data within the registers. To confirm this, the data within the shared memory registers were constantly being changed from one device while the other two devices read that changing data. This process was performed with each device to confirm the accuracy of data transmissions from each instantiation. Additionally, an offset for the address of registers to read and write was used for each device. This was done to confirm that the data and register addresses were dynamic in nature to confirm that the NSL algorithm was working as intended. The result of this testing procedure was a successful validation that the three embedded Modbus RTU modules were working as intended. All data was accurate and readily available to each device, while zero anomalies occurred during this testing phase.

Internal DSP Modbus RTU Testing

The previous paragraphs discussed testing the three embedded Modbus RTU instantiations in the UCB hardware device. To confirm that the internal signals between the FPGA and DSP are working as intended, a new testing configuration was used. This testing configuration utilizes an external “Small UCB,” seen in green in Figure 4.60, to connect with the Code Composer Studio (CCS) software. This software, in conjunction with the Small UCB, allows for the debugging of variable values within the DSP while keeping the serial TI port available.

The DSP operates on code made with the C programming language. To communicate with the FGPA via Modbus RTU, code in C was created by another researcher that was used for this data communication pathway. To confirm the validity of the data processed via embedded Modbus RTU, a two-system testing setup was used. The CCS software component allows for manipulating and viewing variables within the DSP. The second component utilized was the LabVIEW HMI through the UCB’s XPORT connection. To confirm that the internal connection between the FPGA and DSPs is functioning correctly, data was stored and read between the CCS software and the LabVIEW HMI. This testing phase confirmed the functionality of the internal communication pathway between the FPGA and the DSPs.

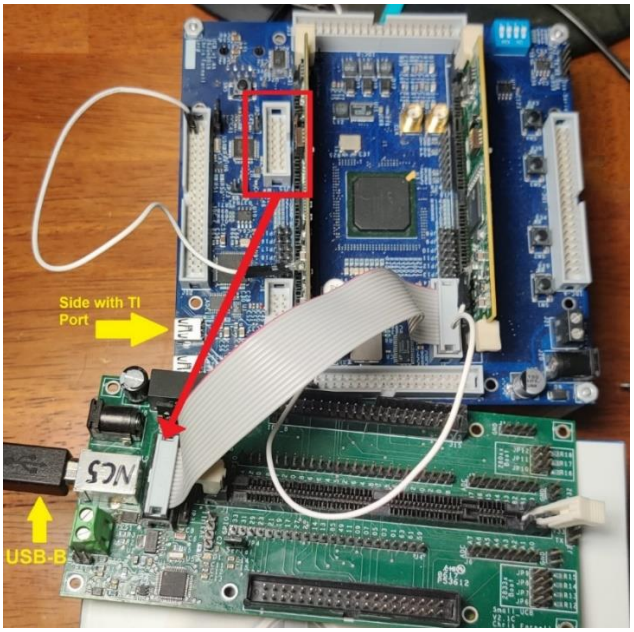


Figure 4.60 – UCB connected with a “Small UCB” (seen in green) through a ribbon cable to provide access to the external debugger for troubleshooting with Code Composer Studio

4.6 – Python Resiliency Testing and Results

Having confirmed the functionality of the integration of the three embedded Modbus RTU instances in the previous testing phase, the next course of action is to conduct resilience testing aimed at identifying the potential failure rate of data transmissions. To perform this test, the NSL algorithm was put through a high iteration process of writing and reading dynamic data consecutively while logging the outcome of these events. This section discusses the setup and actions to perform an exhaustive test on the reliability of the embedded Modbus RTU NSL algorithm.

Timing

One of the potential operating system-specific environmental challenges to consider is the differences in the overall time it takes to service a request between a Linux and Windows-based operating system. In an operating system, an interrupt is a signal indicating that the system needs to stop the current process and process new instructions [37]. The amount of time it takes to process an interrupt depends on many factors, such as specific hardware, software, and type of interrupt. The Linux operating system is known for requiring less overhead and fulfilling software interrupts with a faster response time. In contrast, Windows is known for requiring more overhead to service its more complex features. The beginning of this testing phase addresses this concern by investigating the timing differences between Modbus RTU actions in Python when fulfilled on a Windows device and a Linux device. Additionally, this timing data was needed by another module of the SETO project regarding blockchain development.

To perform this test, a Python program was created using the “MinimalModbus” Python package that loops through many consecutive read and write commands while recording the time to fulfill each action. As expected, the average time to fulfill a Modbus RTU command was higher on the Windows device when compared to the Linux device. The results for the Windows device can be seen in Figure 4.61, while the results for the Linux device can be seen in Figure 4.62. The average time for the Windows and Linux devices to fulfill a Modbus RTU command is approximately 172ms and 161ms, respectively. Additionally, the number of errors received from the “MinimalModbus” package was reported to be zero for each test.

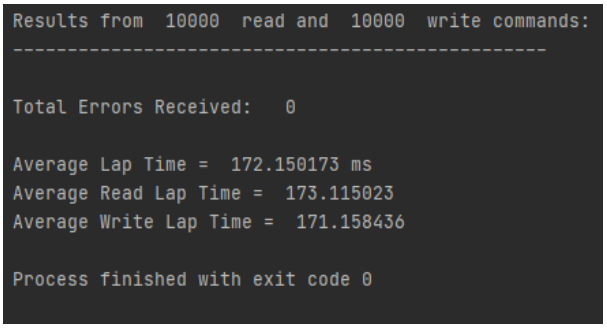


Figure 4.61 – Results from testing the average time to fulfill a Modbus RTU read and write command on a Windows device

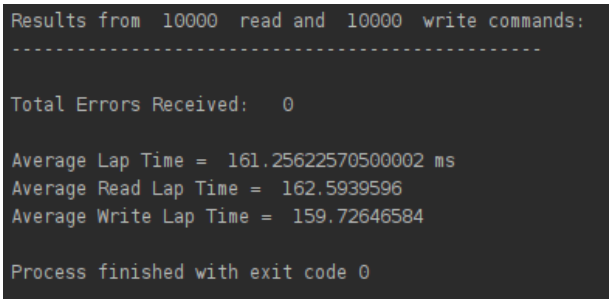


Figure 4.62 - Results from testing the average time to fulfill a Modbus RTU read and write command on a Linux device

Data Accuracy

The next step of this testing phase is to verify the accuracy of stored and received data. To perform this test, the previous Python program was modified to add dynamically changing data in the read and write process. The program operates by writing sixty-four 16-bit register values into the shared memory of the FPGA. Those values are then read back and compared with the previously written values. If these two Python data lists do not match, a variable is incremented to count the occurrences of invalid data. For the next iteration of write and read commands, the data written into the shared memory of the FPGA is first incremented by one at each component of the list. This process is then looped for a total of 10,000 iterations. The results of this test are nominal, and zero occurrences of a failed message or incorrect data occurred, as seen below in Figure 4.63.

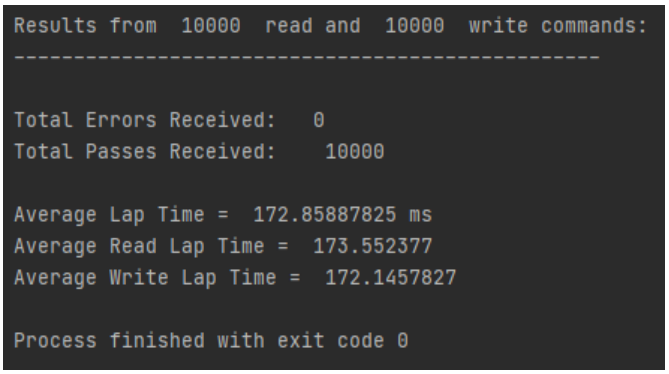


Figure 4.63 - Results from testing the accuracy of data for 10,000 iterations of Modbus RTU read and write commands; the results were nominal, all data passing the accuracy check

4.7 – Bootloader and Hotpatching Testing and Results

The bootloader and hotpatching testing phase are the first attempt at integrating the NSL algorithm with pertinent modules of the SETO project. This integration effort tests the embedded Modbus RTU with the digital twin and firmware validation modules. This section will discuss the testing of the NSL algorithm in a setup that utilizes a LabVIEW HMI with the embedded Modbus RTU communication pathway to boot load inverter controller firmware, assist with performing firmware validation, and access the data of the ANPC inverter digital twin emulation.

This testing phase takes place in the physical system of the UCB. It is facilitated with a LabVIEW HMI that has been modified from a previous LabVIEW HMI created by Dr. Chris Farnell during the preliminary design of the digital twin architecture of the SETO project. The main page of the LabVIEW HMI can be seen below in Figure 4.64. This figure has four sections highlighted in blue, red, yellow, and orange. The blue section represents the configuration settings for the Modbus RTU communication protocol, and the backend of this section can be seen in Figure 4.65. The red section displays the interface to start the bootloading, emulation, and hotpatching. Additionally, this section provides feedback for various status indicators and firmware validation and hotpatching results. The backend for this section can be seen in Figure 4.66. The yellow section displays the configuration settings and status indicators for storing the ANPC inverter controller firmware in the shared memory of the FPGA. The backend that facilitates this process can be seen below in Figure 4.67. The tab that is highlighted in orange at the top is

the datalogger of the HMI that is responsible for displaying the three-phase inverter waveforms of the inverter controller firmware that has been loaded. The datalogger HMI can be seen in Figure 4.70, while the backend for this process can be seen in Figure 4.68.

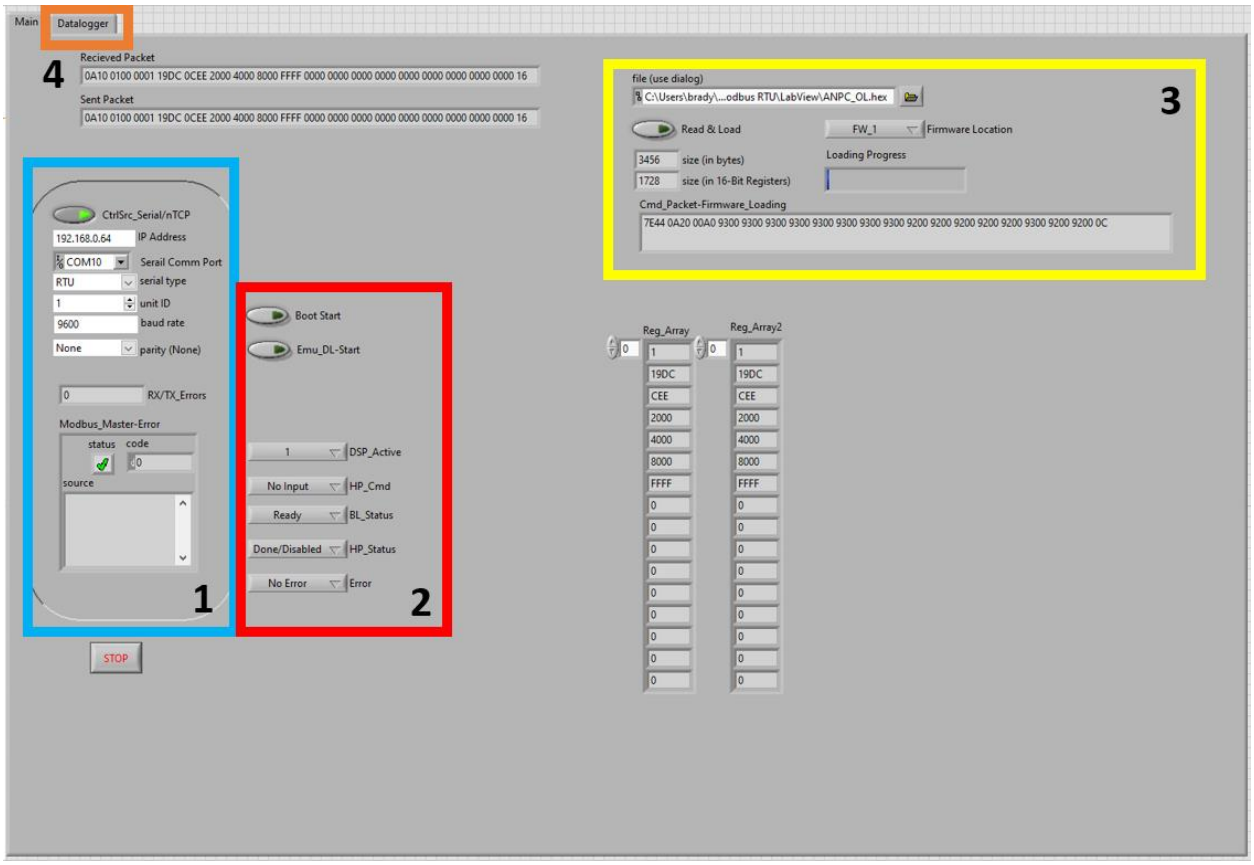


Figure 4.64 – LabVIEW HMI for bootloader and hotpatching features of the SETO project, 1) (blue) configuration settings for Modbus RTU, 2) (red) interface for bootloading, digital twin emulation, hotpatching, and DFTr, 3) (yellow) interface for bootloading of ANPC inverter firmware and status indicator, 4) (orange) datalogger for digital twin ANPC inverter emulation waveforms

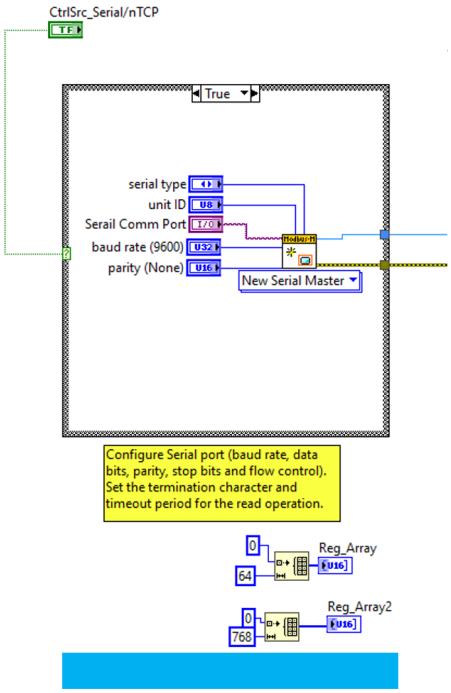


Figure 4.65 – LabVIEW HMI backend of Modbus RTU configuration settings shown in Figure 4.64

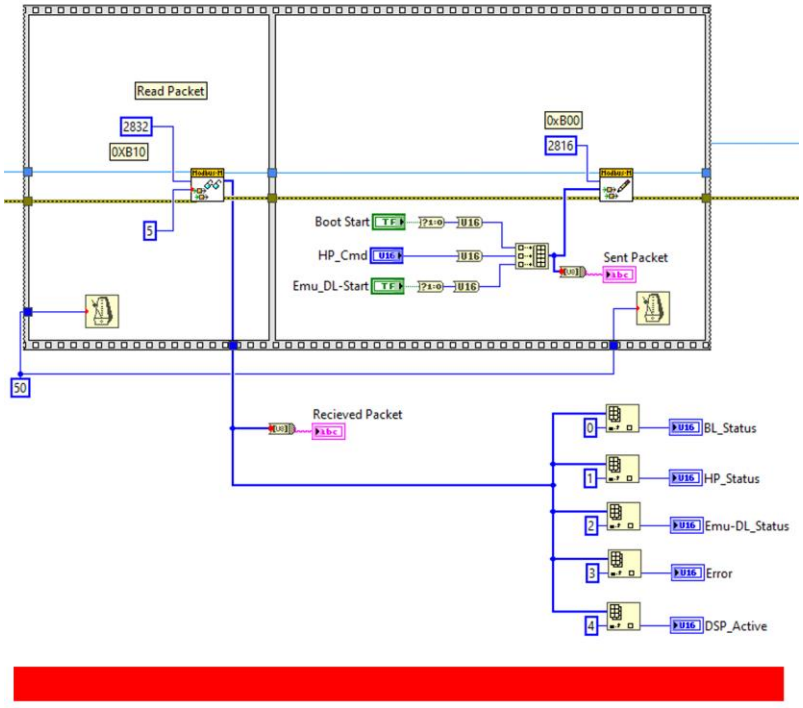


Figure 4.66 - LabVIEW HMI backend of interface and indicators for bootloading, digital twin emulation, hotpatching, and DFTr showed in Figure 4.64

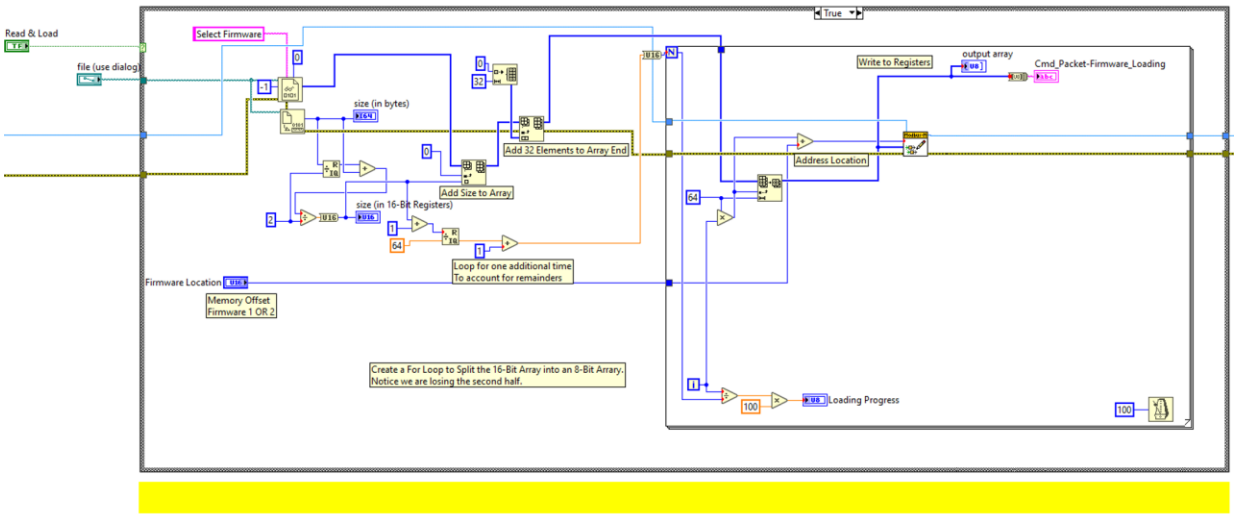


Figure 4.67 - LabVIEW HMI backend of bootloading process of ANPC inverter firmware and status indicator shown in Figure 4.64

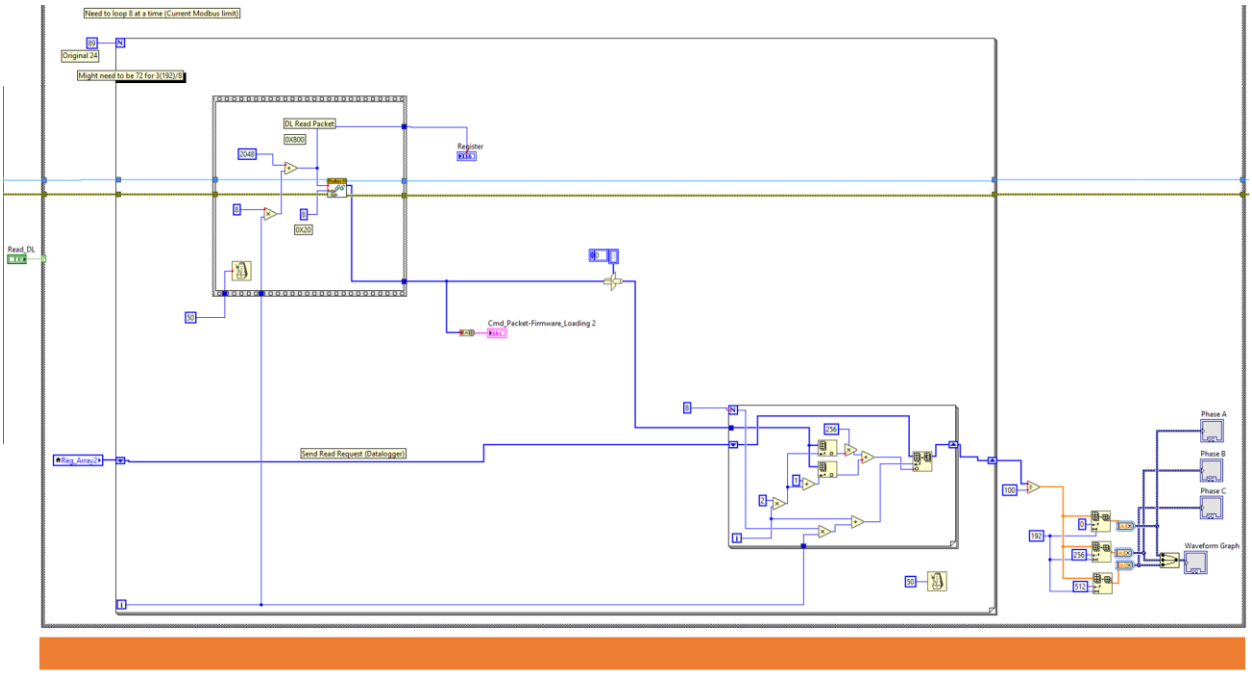


Figure 4.68 - LabVIEW HMI backend of datalogger of the Digital Twin emulation of ANPC inverter shown in Figure 4.64

This testing phase, in the beginning, presented some challenges as it exposed a bug within the coding of the NSL algorithm. Initially, the symptom of this bug appears to be related to a timing issue of serial communication. The initial attempt to establish the communication connection of Modbus RTU communication within LabVIEW would often fail and need to be reattempted numerous times before succeeding. Once the connection was established, the system worked as intended. During this troubleshooting, this symptom was referred to as the “latch on” issue, as the embedded Modbus RTU algorithm needed multiple attempts to establish the initial connection. This timing issue was solved with two different code updates. The first update is adjusting the 3.5-character framing time for clock drift or slight timing deviations. The 3.5-character delay was given an additional 5% timing buffer to ensure the algorithm abides by the minimum required delay. The second update delays the algorithm’s beginning to ensure the reset states and variables have properly settled. This could have been an instance where the metastability of register values after the reset state was not as intended when the NSL algorithm started its process. Once these code updates were made, the embedded Modbus RTU “latched on” as intended and performed nominally.

To test the integration of the embedded Modbus RTU with the hotpatching, bootloading, and DFTr modules of the SETO project, various inverter controller firmware was used. The first step of this testing process was to take an inverter control firmware as a binary file from the local computer hosting the LabVIEW HMI and store it in the shared memory of the FPGA. This step utilizes the LabVIEW HMI backend of Figure 4.67 to perform an iterative action of writing the binary file into the address location for holding inverter controller firmware. The writing process iterates through the binary file by partitioning the data into sixty-four 16-bit register values and sequentially writing them into the shared memory of the FPGA through Modbus RTU write commands. A screenshot of this process in action can be seen during the bootloading process in Figure 4.69. Additionally, as the UCB of the SETO project holds multiple inverter controller firmware, the functionality for various firmware storage locations was tested and verified.

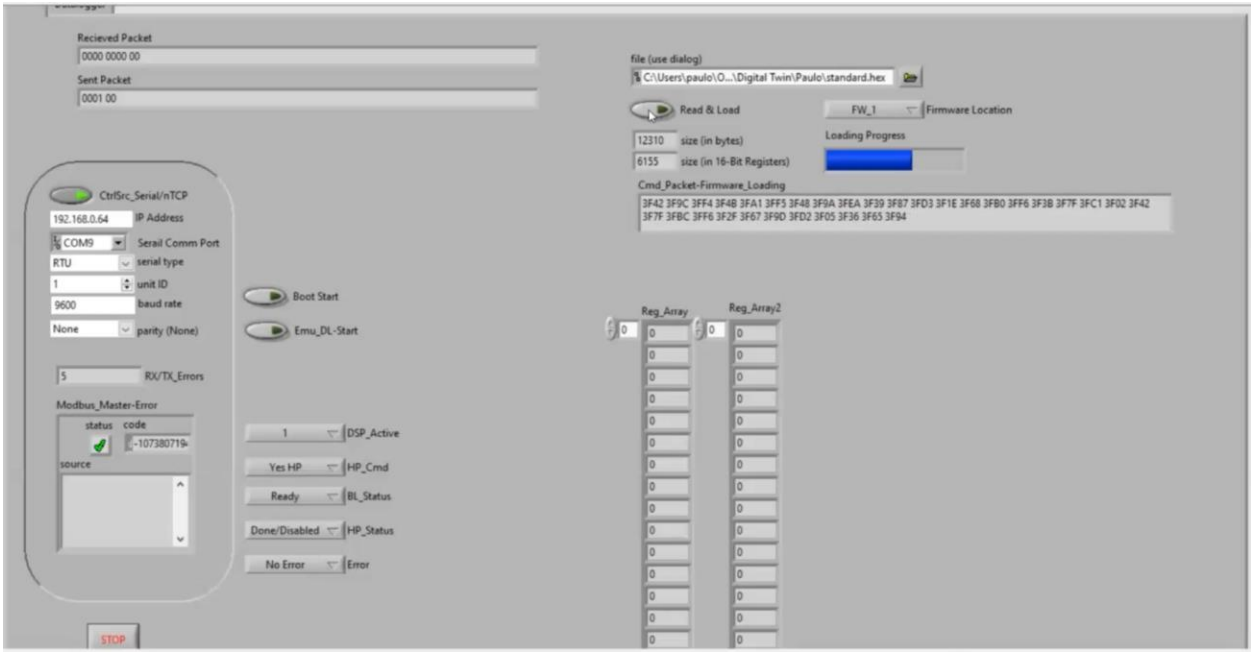


Figure 4.69 – LabVIEW HMI running the bootloader sequence to load inverter controller firmware into the ANPC inverter Digital Twin of the SETO project for the firmware validation of the DFTr module

As the inverter controller firmware has now been successfully stored in the shared memory of the FPGA using the embedded Modbus RTU pathway, the next step is to test the integration of the digital twin emulation process. The embedded Modbus RTU's contribution to this process is initiating the digital twin emulation process and retrieving the resulting data from the shared memory of the FPGA to display within the LabVIEW HMI. First, a Modbus RTU command is sent from the LabVIEW HMI to the digital twin module within the FPGA to initiate the emulation and data-logging process. Once completed, the data is read from the FPGA's shared memory into the LabVIEW HMI datalogger page. The available emulation data from the shared memory of the FPGA is retrieved through a Modbus RTU read command sequentially, sixty-four 16-bit register values at a time. The backend for this process was previously mentioned above in Figure 4.68. The results of this process were nominal, and a screenshot of a resulting emulated ANPC inverter three-phase waveform can be seen below in Figure 4.70.

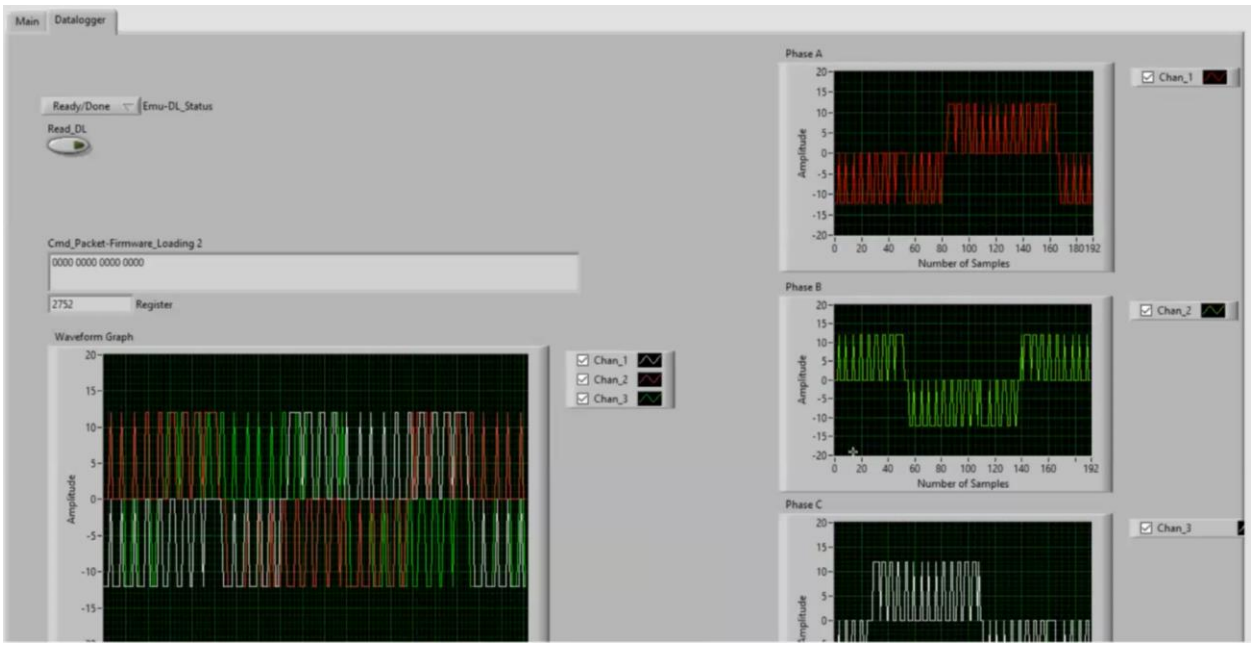


Figure 4.70 – LabVIEW HMI datalogger view of ANPC inverter controller output waveforms of a three-phase system

The final portion of this testing phase is to confirm the integration functionality of the embedded Modbus RTU with the DFTr firmware validation and digital twin hotpatching modules. The embedded Modbus RTU is responsible for sending the start commands and providing the configuration settings from the LabVIEW HMI to the DFTr and hotpatching FPGA modules. To perform the hotpatching, the configuration setting of the hotpatching command is sent through the embedded Modbus RTU communication pathway to the FPGA. Once the hotpatching module receives this command, it performs its function and returns various variable data indicating the status of the hotpatching process. The resulting status of the hotpatching module is returned to the LabVIEW HMI via a Modbus RTU read request. Regarding the DFTr module, this FPGA module is run during the bootloading of inverter controller firmware. The resulting bootloading status and any potential errors found during the firmware validation process are returned to the LabVIEW HMI via a Modbus RTU read request. A screenshot of the resulting status indicators of these processes can be seen below in Figure 4.71, where the DFTr found a short circuit scenario within the inverter controller firmware and has defaulted to loading a known good backup firmware. The algorithm for loading backup firmware, DFTr, and hotpatching is not discussed in

this thesis work. These modular components within the SETO project require integration with Modbus RTU but are not part of this work.

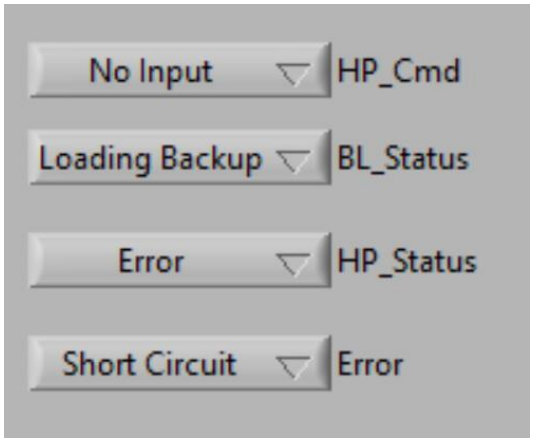


Figure 4.71 – LabVIEW HMI interface output of results of DFTr SETO module indicating a short circuit error exists in the inverter controller firmware

4.8 – Resource Utilization

One of the primary goals of this work is to create an embedded Modbus RTU implementation within a low-cost FPGA with limited space and efficiently use the limited available resources. To correctly measure this metric, a baseline for resource utilization of a commercial off-the-shelf implementation must be established. A primary provider of embedded system architecture devices and solutions is Intel's Altera line of products. The Nios II embedded processor architecture by Altera provides the processing requirements to fulfill the Modbus RTU communication protocol functionality by compiling a Modbus RTU implementation coded using the C programming language. This setup also requires the C code containing the Modbus RTU protocol algorithm to be stored on the FPGA device.

Within low-cost FPGAs, there are limited RAM and flash storage resources available. The MachXO2-7000HC has 240 kilobits of RAM and 256 kilobits of flash storage. In contrast, an available open-source C code implementation of Modbus RTU requires 36 kilobytes of storage. This storage requirement is orders of magnitude larger than available on the FPGA of this work. To store the required Modbus RTU C code for use with an embedded processor in the MachXO2-7000HC FPGA, external storage is required. To utilize this external storage, transmitting data from the external store into the

embedded processor within the FPGA requires a method of communication. A communication protocol such as SPI can be used with an Execute-in-Place (XIP) method to execute code directly from the external storage [38]. A diagram of this setup utilizing external storage components and implementation methods can be seen in

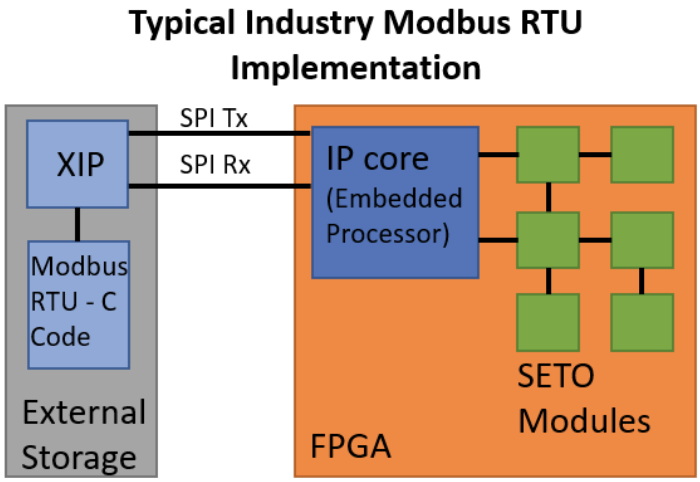


Figure 4.72 – Diagram of a typical industry method for implementing Modbus RTU within an embedded system using external storage to house the related Modbus RTU C code and a method of retrieving the code to be executed from the external storage

The most relevant technical requirement to measure the impact of resource allocation within the FPGA is that of the LUTs previously mentioned in Chapter 2. This is due to the LUTs being the most constrained resource of the FPGA for the embedded SETO modules. To implement the Nios II embedded processor architecture, a minimum of 2800 LUTs is required within the FPGA [39]. The embedded Modbus RTU presented in this thesis work requires 2561 LUTs to store all three modular communication pathways for the RPi, DSP, and LabVIEW connections. A diagram comparing the typical industry method of implementation with the embedded Modbus RTU of this work can be seen in Figure 4.73. The design summary containing the resource utilization of this implementation within the UCB's FPGA can be seen in Figure 4.74, where the design without any Modbus RTU modules is on the left, and the design with the three Modbus RTU modules is on the right. The difference in LUTs between these two summaries can be calculated as the stated 2561 LUTs. The presented implementation is approximately 9% smaller than a commercial off-the-shelf application. Additionally, this difference in utilized resources does not account for

the space required to host the C programming language code of the Modbus RTU protocol for the Nios II embedded processor architecture.

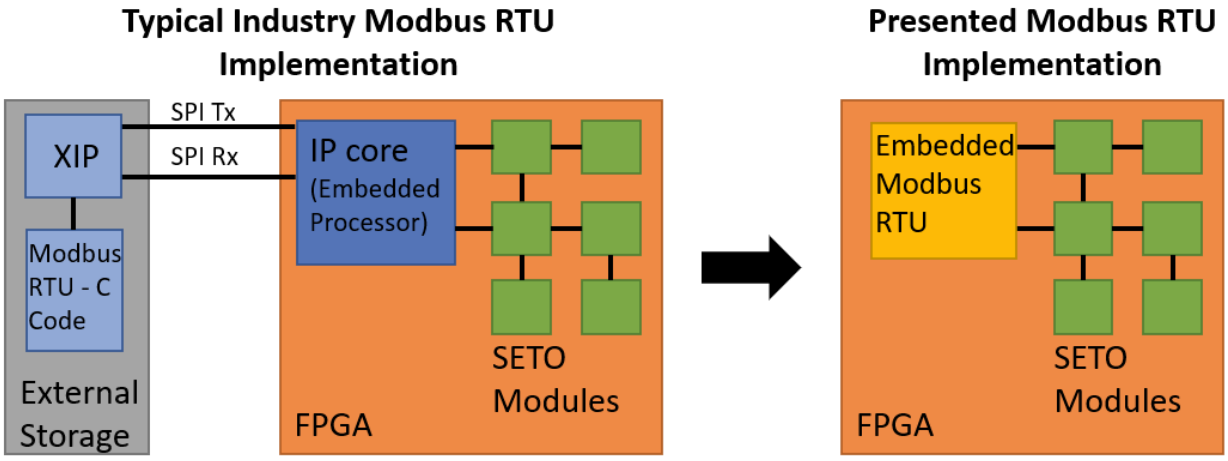


Figure 4.73 – Comparison of typical industry embedded processor-based Modbus RTU implementation and the embedded Modbus RTU presented in this work

No Modbus

```
Design Summary
   Number of registers:   2726 out of  7869 (35%)
      PFU registers:          2719 out of  6864 (40%)
      PIO registers:             7 out of  1005 (1%)
   Number of SLICEs:      1889 out of  3432 (55%)
      SLICEs as Logic/ROM:   1889 out of  3432 (55%)
      SLICEs as RAM:            0 out of  2574 (0%)
      SLICEs as Carry:        834 out of  3432 (24%)
   Number of LUT4s:        3735 out of  6864 (54%)
      Number used as logic LUTs:        2067
      Number used as distributed RAM:      0
      Number used as ripple logic:      1668
      Number used as shift registers:      0
   Number of PIO sites used: 99 + 4(JTAG) out of 335 (31%)
   Number of block RAMs:  11 out of 26 (42%)
   Number of GSRs:        0 out of 1 (0%)
   EFB used :         No
   JTAG used :        No
   Readback used :    No
   Oscillator used :  Yes
   Startup used :     No
   POR :              On
   Bandgap :          On
   Number of Power Controller:  0 out of 1 (0%)
   Number of Dynamic Bank Controller (BCINRD):  0 out of 6 (0%)
   Number of Dynamic Bank Controller (BCLVDSO):  0 out of 1 (0%)
   Number of DCCA:  0 out of 8 (0%)
   Number of DCMA:  0 out of 2 (0%)
   Number of PLLs:  1 out of 2 (50%)
   Number of DQSDLLs:  0 out of 2 (0%)
   Number of CLKDIVC:  0 out of 4 (0%)
   Number of ECLKSYNCA:  0 out of 4 (0%)
   Number of ECLKBRIDGECS:  0 out of 2 (0%)
```

Modbus x3

```
Design Summary
   Number of registers:   4403 out of  7869 (56%)
      PFU registers:          4392 out of  6864 (64%)
      PIO registers:            11 out of  1005 (1%)
   Number of SLICEs:      3230 out of  3432 (94%)
      SLICEs as Logic/ROM:   3230 out of  3432 (94%)
      SLICEs as RAM:            0 out of  2574 (0%)
      SLICEs as Carry:       1160 out of  3432 (34%)
   Number of LUT4s:        6296 out of  6864 (92%)
      Number used as logic LUTs:        3976
      Number used as distributed RAM:      0
      Number used as ripple logic:      2320
      Number used as shift registers:      0
   Number of PIO sites used: 103 + 4(JTAG) out of 335 (32%)
   Number of block RAMs:  20 out of 26 (77%)
   Number of GSRs:        0 out of 1 (0%)
   EFB used :         No
   JTAG used :        No
   Readback used :    No
   Oscillator used :  Yes
   Startup used :     No
   POR :              On
   Bandgap :          On
   Number of Power Controller:  0 out of 1 (0%)
   Number of Dynamic Bank Controller (BCINRD):  0 out of 6 (0%)
   Number of Dynamic Bank Controller (BCLVDSO):  0 out of 1 (0%)
   Number of DCCA:  0 out of 8 (0%)
   Number of DCMA:  0 out of 2 (0%)
   Number of PLLs:  1 out of 2 (50%)
   Number of DQSDLLs:  0 out of 2 (0%)
   Number of CLKDIVC:  0 out of 4 (0%)
   Number of ECLKSYNCA:  0 out of 4 (0%)
   Number of ECLKBRIDGECS:  0 out of 2 (0%)
```

Figure 4.74 - Resource allocation comparison within the FPGA of the project without (left) and with (right) the three embedded Modbus RTU instantiations

Chapter 5

Conclusions and Future Work

5.1 – Summary of Conclusions

The industrial control systems field widely embraces the Modbus RTU communication protocol due to its reliability and ease of implementation. It offers flexibility in operating parameters and supports multiple formats and hardware infrastructures. Many devices like PLCs, HMIs, IoT, and OT use Modbus RTU, requiring a centralized communication process to connect modular systems. An FPGA can be an embedded central routing fabric to facilitate this communication.

Embedded systems are expected to be versatile enough to interface with various devices while being physically small to require less “real-estate” space and resource-efficient to minimize costs. To establish a baseline standard for communication among systems, a typical method of implementing the Modbus RTU communication protocol in an embedded environment is to use IP cores as embedded processors within the FPGA. The VHDL hardware description language allows hardware control at the logic gate and signal level, providing precision over creating the logical circuit within the FPGA. This precision can be used to create an efficient embedded Modbus RTU implementation with respect to the previously stated challenges.

This work has presented the Implementation of the Modbus RTU communication protocol using VHDL programming within an FPGA to establish a standardized and embedded serial communication pathway that assists with streamlining research efforts and increasing efficiency. This implementation requires fewer resources than typical communication protocol implementations, reducing hardware requirements for effective research efforts. In conclusion, the two challenges within the cyber-physical research field of efficient collaborative systems and cost-effective solutions have been effectively addressed by this presented embedded Modbus RTU implementation.

5.2 – Summary of Future Work

While this presented work has successfully met all the milestones to fulfill its role of providing the communication routing fabric for the SETO project, two potential points of improvement have been identified to continue this work. The first improvement is an increase in the communication configuration of the baud rate. The system currently operates at a baud rate of 9,600 but could be improved to a baud rate of 115,200, increasing the system throughput. The second potential upgrade is a proof-of-concept modification that would add encryption and message authentication features to the data requested from slave devices. The remainder of this section will discuss these future works in further detail, discuss current steps taken to accomplish these milestones and provide the author's perspective on the next steps to accomplish these tasks.

Increase Baud Rate to 115,200

While the operational baud rate of 9600 bits-per-second works effectively to provide communication capabilities for this project, the throughput can be improved while potentially further reducing the overhead-related resource allocation by upgrading to a system capable of a higher baud rate. Within serial communication, some common baud rates are 9,600, 19,200, and 115,200 bits-per-second. The embedded Modbus RTU NSL algorithm can be modified to increase the baud rate from 9,600 to 115,200, thus decreasing the time it takes to send and receive messages. This will result in the entire system of the UCB spending less overhead on servicing the communication protocol with respect to time.

While this increase in the transmission rate is not in the original scope of work, some efforts have been put into addressing this potential improvement. The author's limited time and resources did not allow this enhancement to be completed. Currently, the bus interface for serial communication allows for scaling the baud rate by using a global VHDL variable. Upon adjusting the system from a baud rate of 9,600 to 115,200, the embedded Modbus RTU functioned as expected within the ModelSim simulations and the development board with XCTU. The preliminary testing of this higher baud rate failed when testing within the UCB with a LabVIEW HMI for troubleshooting. The researcher was not able to identify the issue with their available time. If one was to continue this work, one should first investigate potential

timing and framing issues that may become apparent at the higher transmission rate. Additionally, there could be an additional hardware-related challenge within the UCB that is not preset within the development board.

Encryption and Message Authentication

An additional enhancement that was preliminarily explored is the proof-of-concept for the encryption and message authentication of Modbus RTU for the responses of read commands. This would allow data being received from the shared memory of the FPGA to be encrypted and authenticated using a set of keys known only to authorized users. Modbus is not traditionally a form of secure messaging as the entire message contains plain text, and the CRC is only intended for message integrity and not message security. To achieve this cyber-security focused improvement, an embedded processor architecture and changes to the VHDL code could be used in conjunction to perform the data encryption and message authentication before sending responses to read commands.

The author, in conjunction with Dr. Qinghua Li's research team at the University of Arkansas, determined for this proof-of-concept effort that the AES-128 encryption model and the SHA-256 message authentication model should be used. Like the commercial off-the-shelf embedded processor architecture of the Altera Nios II processor, IP cores exist for AES-128 and SHA-256 encryption and message authentication, respectively. To implement this, it would require reworking the response portion of the Modbus RTU algorithm, originally seen in Figure 3.40, to feed the data to be encrypted and authenticated into these IP cores before sending out the response message. A diagram of the NSL algorithm with the modified addition of these IP cores can be seen below in Figure 5.75 and is colored in black. This will effectively provide a Modbus RTU message that has the data portion of the packet encrypted and authenticated, where the decryption and authentication will be performed on the client side of the communication. The original decryption and authentication of this data were intended to be performed on a Raspberry Pi using the appropriate Python packages.

This work has been postponed indefinitely due to the limited time and resources of the associated researchers. Additionally, this system would require an FPGA with additional available resources. Adding these encryption and message authentication IP cores increased the project size of the SETO UCB

program to exceed the available LUTs of the FPGA to a LUT utilization of approximately 125%. If one is to continue this work, the team at NCREPT is creating an improved version of the UCB that could support the addition of these IP cores.

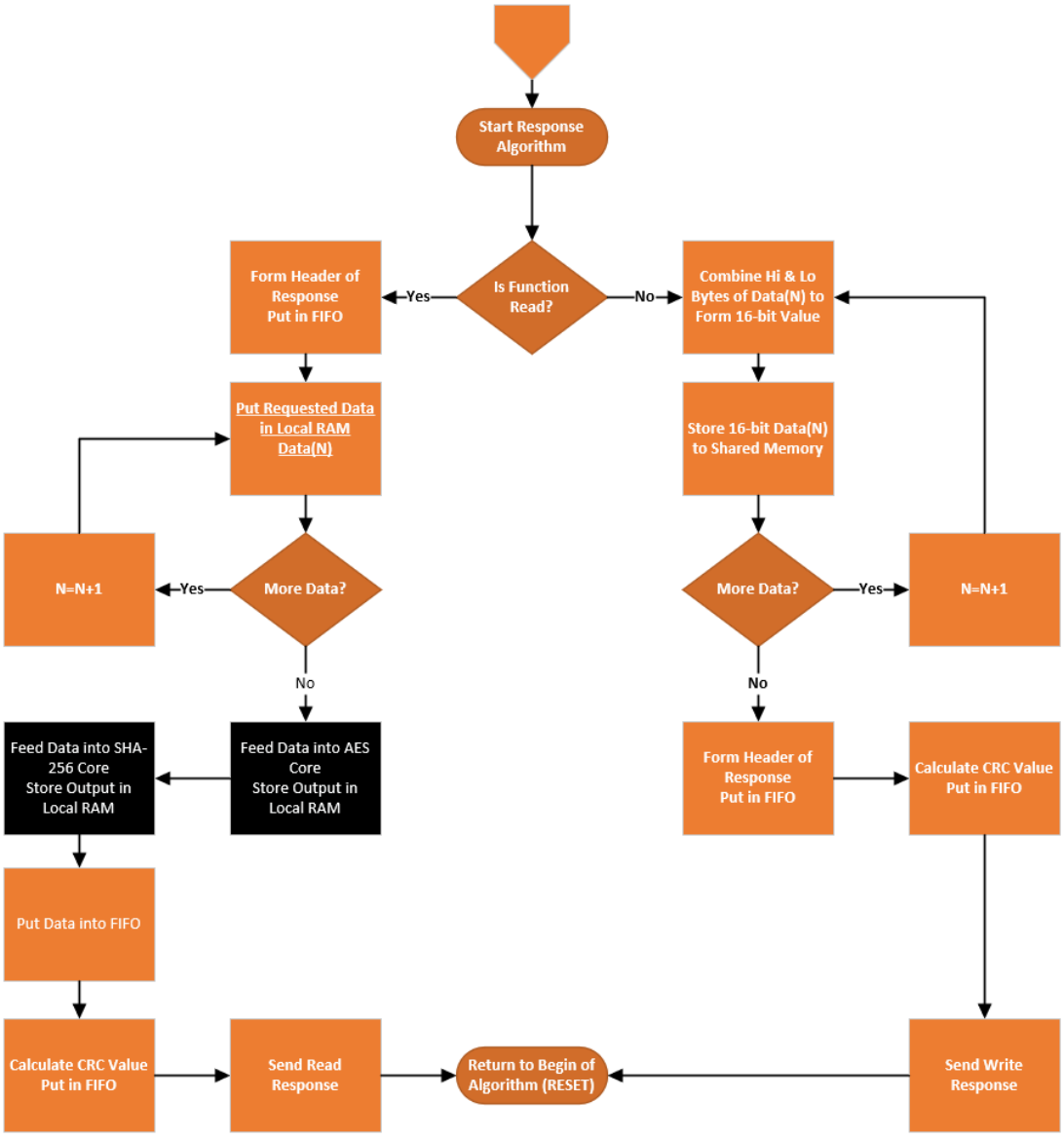


Figure 5.75 - State diagram of the next-state-logic algorithm used for the processing and fulfillment of the Modbus RTU command response received by the slave device modified with the AES-128 encryption and SHA-256 message authentication IP cores (seen in black) incorporated into the algorithm

5.3 – Discussion on the Limitations of Low-Cost FPGAs

While one of the main goals of this project was to develop an embedded Modbus RTU application suitable for a low-cost FPGA, there are some potential drawbacks to utilizing these low-cost integrated circuits. At the time of writing, a single MachXO2-7000HC FPGA can be purchased for approximately $23 and was initially released in 2012. While this FPGA could fit multiple cyber-physical security-focused modules to be used for the controller architecture of a solar inverter, there are limitations to the capabilities of this device. Regarding the capabilities of different FPGA manufacturers, Lattice products are typically known for being equipped with the least available resources compared with other manufacturers, such as Altera and Xilinx. Excluding market anomalies, the price of an FPGA typically goes together with the device's capabilities, as seen in Figure 5.76.

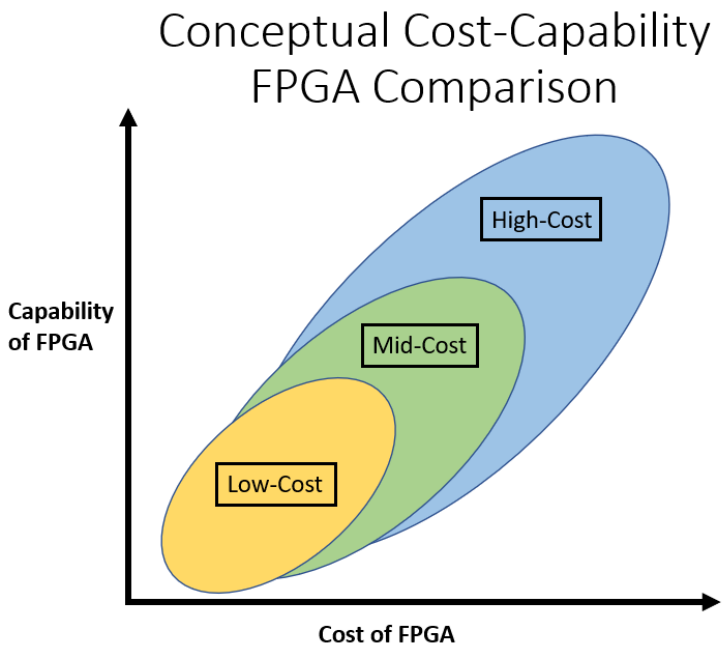


Figure 5.76 – Conceptual comparison between the capabilities of an FPGA with respect to cost

Previously mentioned in the future work subsection was the proof-of-conceptual project to encrypt outgoing Modbus RTU messages while adding a message authentication hash to validate the message contents. As this additional code required more LUTs than the Lattice FPGA had available, the research efforts for this addition were discontinued. The encryption and message authentication functionality, along

with the other SETO modules within the FPGA, would require approximately 8,125 LUTs, while the Lattice MachXO2-7000HC only contains 6,468 LUTs. Comparatively, the MachXO3D-9400HC from 2019 can be purchased for $22 and contains 9400 LUTs. A comparison of the available resources between these two Lattice FPGA models that were released approximately seven years apart can be seen below in Table 4.

Table 4 – Comparison of resources between Lattice's MachXO2-7000HC and MachXO3D-9400HC

Resource	Lattice MachXO2-7000HC (2012)	Lattice MachXO3D-9400HC (2019)	Delta (%)
Look Up Tables (LUTs)	6864	9400	+37%
EBR SRAM (kbits)	240	432	+80%
Flash Memory (kbits)	256	2700 (up to)	+955%

Based on the available resources of the MachXO3D-9400HC, the encryption and message authentication features previously mentioned could be implemented using this FPGA along with the original SETO modules. While the cost difference between these two devices is negligible at the time of writing, the capability between them is significantly noticeable. This is due to the improvements in available technologies between the release dates of the two models. While the financial benefits of a low-cost FPGA allow for the application and development of cost-effective solutions, additional benefits can be obtained from utilizing more capable and contemporary hardware. At this time, researchers at the University of Arkansas are developing a second version of the UCB equipped with this MachX03D-9400HC FPGA. If one seeks to continue securing and authenticating the normally non-secure Modbus RTU message in conjunction with additional FPGA-based system modules, this improved UCB could support such functionality.

References

[1] IEA, "Renewables 2021 Analysis and forecats to 2026," IEA, 2021.

[2] M. I. Henderson, D. Novosel and M. L. Crow, "Electric Power Grid Modernization Trends, Challenges, and Opportunities," November 2017. [Online]. Available: https://www.ieee.org/content/dam/ieee-org/ieee/web/org/about/corporate/ieee-industry-advisory-board/electric-power-grid-modernization.pdf. [Accessed 9 February 2023].

[3] L. Che, X. Liu, T. Ding and Z. Li, "Revealing Impacts of Cyber Attacks on Power Grids Vulnerability to Cascading Failures," *IEEE Transactions on Circuits and Systems II: Express Briefs,* vol. 66, no. 6, pp. 1058 - 1062, 2019.

[4] R. Walton, "Infrastructure bill allots billions to enhance grid's cyber defenses, creates new authority for Homeland Security," Utility Dive, 8 November 2021. [Online]. Available: https://www.utilitydive.com/news/infrastructure-bill-allots-billions-to-enhance-grids-cyber-defenses-creat/609626/. [Accessed 9 February 2023].

[5] E. Monmasson and M. N. Cirstea, "FPGA Design Methodology for Industrial Control Systems - A Review," *IEEE Transactions on Industrial Electronics,* vol. 54, no. 4, pp. 1824-1842, 2007.

[6] K. Wang, D. Peng, L. Song and H. Zhang, "Implementation of Modbus Communication Protocol Based on ARM Cortex-M0," in *IEEE International Conference on System Science and Engineering (ICSSE)*, Shanghai, 2014.

[7] D. Peng, H. Zhang, J. Weng, H. Li and F. Xia, "Design and Realization of Modbus Protocol Based on Embedded Linux System," in *International Conference of Embedded Software and Systems Symposia*, Beijing, 2008.

[8] D. Peng, H. Zhang, J. Weng, H. Li and X. Fei, "Design and Development of Modbus/RTU Master Monitoring System Based on Embedded PowerPC Platform," in *IEEE International Symposium of Industrial Electronics*, Seoul, 2009.

[9] W. Yongliang, C. Jiejie, L. Xinqi, H. Yunqi, L. Dengcheng, Y. Daliang and C. Ji, "Design of Environmental Control System Based on Embedded Modbus," in *Chinese Control And Decision Confernece (CCDC)*, Nanchang, 2019.

[10] M. Urbina, A. Astarloa, J. Lazaro, U. Bidarte, I. Villalta and M. Rodriguez, "Cyber-Physical Production System Gateway Based on a Programmable SoC Platform," *IEEE Access,* vol. 5, pp. 20408-20417, 2017.

[11] SMA-America, "Sunny Highpower PEAK3," SMA-America, 19 September 2022. [Online]. Available: https://www.sma-america.com/products/solarinverters/sunny-highpower-peak3. [Accessed 30 January 2023].

[12] N. Rodríguez-Pérez, J. M. Domingo, G. L. López and V. Stojanovic, "Scalability Evaluation of a Modbus TCP Control and Monitoring System for Distributed Energy Resources," in *2022 IEEE PES Innovative Smart Grid Technologies Conference Europe (ISGT-Europe)*, Novi Sad, 2022.

[13] CONTEC, "Serial Communication Basic Knowledge - RS-232C / RS-422A / 485," [Online]. Available: https://www.contec.com/support/basic-knowledge/daq-control/serial-communicatin/. [Accessed 30 January 2023].

[14] T. Sharma, "RS232 Serial Communication Protocol: Basics, Working & Specifications," Circuit Digest, 1 January 2018. [Online]. Available: https://circuitdigest.com/article/rs232-serial-communication-protocol-basics-specifications. [Accessed 30 January 2023].

[15] J. Blom, "Serial Communication," sparkfun, [Online]. Available: https://learn.sparkfun.com/tutorials/serial-communication/wiring-and-hardware. [Accessed 31 January 2023].

[16] E. Peña and M. G. Legaspi, "UART: A Hardware Communication Protocol Understanding Universal Asynchronous Receiver/Transmitter," *Analog Dialogue,* vol. 54, 2020.

[17] "An Introduction to Modbus RTU," RTA Automation, [Online]. Available: https://www.rtautomation.com/technologies/modbus-rtu/. [Accessed 1 February 2023].

[18] M. Barr, "CRC Series, Part 2: CRC Mathematics and Theory," *Embedded Systems Programming,* no. December 1999, pp. 47-54, December 1999.

[19] F. L. Marques, "Graduation work: creation of a low cost automation system using open source technologies and embedded systems," August 2015. [Online]. Available: https://www.researchgate.net/publication/280777956_Trabalho_de_graduacao_criacao_de_um_sistema_de_automacao_de_baixo_custo_utilizando_tecnologias_open_source_e_sistemas_embarcados_rev. [Accessed 22 February 2023].

[20] Modbus Organization, Inc., "MODBUS over Serial Line Specification and Implementation Guide," 20 December 2006. [Online]. Available: https://modbus.org/docs/Modbus_over_serial_line_V1_02.pdf. [Accessed 10 February 2023].

[21] "Detailed description of the Modbus TCP protocol with command examples," IPC2U, [Online]. Available: https://ipc2u.com/articles/knowledge-base/detailed-description-of-the-modbus-tcp-protocol-with-command-examples/. [Accessed 7 February 2023].

[22] C. Farnell, E. Soria, J. Jackson and H. A. Mantooth, "Cyber Protection of Grid-Connected Devices Through Embedded Online Security," in *IEEE Design Methodologies Conference (DMC)*, 2021.

[23] Advanced Micro Devices, Inc, "What is an FPGA?," AMD XILINX, [Online]. Available: https://www.xilinx.com/products/silicon-devices/fpga/what-is-an-fpga.html#:~:text=Field%20Programmable%20Gate%20Arrays%20(FPGAs,or%20functionality%20requirements%20after%20manufacturing.. [Accessed 13 February 2023].

[24] Invent Logics, "FPGA Architecture," All About FPGA, 16 April 2014. [Online]. Available: https://allaboutfpga.com/fpga-architecture/. [Accessed 13 February 2023].

[25] Lattice Semiconductor, "MachX02," Lattice Semiconductor, 23 March 2012. [Online]. Available: https://www.latticesemi.com/Products/FPGAandCPLD/MachXO2. [Accessed 13 February 2023].

[26] S. H. L., "Purpose of Internal Functionality of FPGA Look-Up Tables," All About Circuits, 9 November 2017. [Online]. Available: https://www.allaboutcircuits.com/technical-articles/purpose-and-internal-functionality-of-fpga-look-up-tables/. [Accessed 14 February 2023].

[27] J. P. Nicolle, "FPGA software 5 - FPGA synthesis and place-and-route," fpga4fun, [Online]. Available: https://www.fpga4fun.com/FPGAsoftware5.html#:~:text=Place-and-route. [Accessed 14 February 2023].

[28] D. S. Arar, "What is VHDL? Getting started with Hardware Description Language for Digital Circuit Design," All About Circuits, 29 December 2017. [Online]. Available:

https://www.allaboutcircuits.com/technical-articles/hardware-description-langauge-getting-started-vhdl-digital-circuit-design/. [Accessed 14 February 2023].

[29] Xilinx, "Modeling Concepts," 2013. [Online]. Available: https://www.xilinx.com/content/dam/xilinx/support/documents/university/ISE-Teaching/HDL-Design/14x/Nexys3/VHDL/docs-pdf/VHDL-Lab1.pdf. [Accessed 14 February 2023].

[30] Buzztech, "VHDL Modelling Sytles: Behavioral, Dataflow, Structural," Buzztech, 2018. [Online]. Available: https://buzztech.in/vhdl-modelling-styles-behavioral-dataflow-structural/. [Accessed 15 February 2023].

[31] Intel, "Implementing State Machines (VHDL)," Intel, 2013. [Online]. Available: https://www.intel.com/content/www/us/en/programmable/quartushelp/13.0/mergedProjects/hdl/vhdl/vhdl_pro_state_machines.htm. [Accessed 15 February 2023].

[32] R. Awati, "Intellectual Propety Core (IP core)," TechTarget, November 2022. [Online]. Available: https://www.techtarget.com/whatis/definition/IP-core-intellectual-property-core#:~:text=An%20intellectual%20property%20core%20(IP%20core)%20is%20a%20functional%20block,circuit%20(IC)%20layout%20design.. [Accessed 15 February 2023].

[33] M. Saikaly, "Common Bus Architecture Throughput," June 2007. [Online]. Available: https://www.ti.com/lit/an/spraan7/spraan7.pdf. [Accessed 20 February 2023].

[34] Texas Instruments, "RS-232 Frequently Asked Questions," 2022. [Online]. Available: https://www.ti.com/lit/pdf/slla544. [Accessed February 16 2023].

[35] National Instruments, "Speed, Rates, Times, Delays: Data Link Parameters for CSE 461," 24 June 2022. [Online]. Available: https://www.ni.com/docs/en-US/bundle/labview-fpga-module/page/lvfpgaconcepts/registers.html. [Accessed 24 February 2023].

[36] Altera, "Understanding Metastability in FPGAs," 2009. [Online]. Available: https://cdrdv2-public.intel.com/650346/wp-01082-quartus-ii-metastability.pdf. [Accessed 27 February 2023].

[37] R. Awati, "What is an interrupt?," Tech Target, July 2022. [Online]. Available: https://www.techtarget.com/whatis/definition/interrupt. [Accessed 3 March 2023].

[38] Microchip, "Execute-In-Place (XIP) with QSPI on Cortex-M7 MCUs Using MPLAB Harmony v3," 2020. [Online]. Available: https://ww1.microchip.com/downloads/en/Appnotes/Execute-In-Place%20_with_QSPI_on_%20Cortex-M7_MCUs_Using_MPLAB_Harmony_v3_DS00003443A.pdf. [Accessed 28 April 2023].

[39] Altera, "AN 717: Nios II Gen2 Hardware Development Tutorial," 22 September 2014. [Online]. Available: https://www.intel.com/content/www/us/en/docs/programmable/683615/current/software-and-hardware-requirements.html. [Accessed 6 March 2023].

[40] "OSI Model," imperva, [Online]. Available: https://www.imperva.com/learn/application-security/osi-model/. [Accessed 2 February 2023].

Appendices

Appendix A: Code

A-1 Modbus RTU Algorithm: FPGA VHDL Code

```
----------------------------------------------------------------------------------
-- Company:  University of Arkansas (NCREPT)
-- Engineer: Brady McBride
--
-- Create Date:                6Jun2022
-- Design Name:                Digital_Twin_UART
-- Module Name:                Digital_Twin_UART - Behavioral
-- Project Name:               Bus Interface for Modbus RTU
-- Target Devices:             LCMXO2-7000HC-4FG484C (UCB v1.3a)
-- Tool versions:              Lattice Diamond_x64 Build 3.10.2.115.1
--
-- Description:
-- This module provides an RS232 UART interface which allows access to the memory locations of the
device.
-- Default configuration is 1 start bit, 1 stop bit, no parity, and 9600bps.
-- This module uses the Common Bus Architecture.
-- This module is based off the serial communication work of Dr. Chris Farnell - cfarnell@uark.edu
--
---- Modbus RTU Communications:
-- In the project we will construct packets to facilitate Modbus RTU communication between the slave
device and a master device.
-- Using packets we can implement operational commands and check sums which allow for easily
implementing a user GUI.
-- You may use various Serial Interface programs to communicate with this including the old version of X-
CTU (Version 5.2.8.6),
-- Termite, HypterTerminal, or the Custom LabVIEW Interface created for this project (LabVIEW-
CommEx_Bus_Interface_v1.1b_9Jun2019).
-- A key to remember is that you will need to create packets in Hex-Mode not the standard ASCII-Mode.
-- All packets will begin with a the device address and end with a CheckSum.
-- The CheckSum is the CRC-16 Modbus calculation. A useful calculator can be found here:
https://crccalc.com/
-- Be sure to click the "CRC-16" button at the top and then click the "Modbus" line item to single out this
calculation.
-- This algorithm provides use for the Modbus RTU reading and writing of holding register data types.
-- All reads and writes are configured to use the "multiple holding register" access commands.
-- Read multiple holding registers.      Functional code:      0x03
-- Write multiple holding registers.     Functional code:      0x10
--
-- The following examples are from:      https://ipc2u.com/articles/knowledge-base/modbus-rtu-made-
simple-with-detailed-descriptions-and-examples/
--
--
--
---Modbus RTU Read Packet (OP_ID = 0x03)
-- The Register Read Command Packet is used to provide register data internal to the CPLD to the
requesting master device.
-- The following example breaks down a read request packet.
-- The packet below writes 7 16-bit registers starting at register 0x0100; Values listed below.
--
```

```
-- Device  | Functional | Start   | Number of | Checksum
-- Address | Code       | Address | Registers | CRC
-- 0x11    | 0x03       | 0x006B  | 0x0003    | 0x7687
-- 0x1103006B00037687
--
-- An example of a response for this read command is below
--
-- Device Address | Functional Code | Number Bytes More | Register Data  | Checksum CRC
-- 0x11                          | 0x03           | 0x06              | 0xAE4156524340 | 0x49AD
-- 0x110306AE415652434049AD
--
--
--
--
--
--
--
---Modbus RTU Write Packet (OP_ID = 0x10)
-- The Register Write Command Packet is used to update registers internal to the CPLD.
-- The following example breaks down a write request packet.
-- The packet below writes 7 16-bit registers starting at register 0x0100; Values listed below.
--
-- Device Address | Functional Code | Start Address | Number of Registers | Number of Bytes More |
Register Data | Checksum CRC
-- 0x11          | 0x10            | 0x0001        | 0x0002              | 0x04                | 0x000A0102    | 0xC6F0
-- 0x11100001000204000A0102C6F0
--
-- An example of a response for this write command is below
--
-- Device Address | Functional Code | Starting Address | Number of Registers | Checksum CRC
-- 0x11          | 0x10            | 0x0001           | 0x0002              | 0x1298
--m 0x1110000100021298
--
--
--
--
--
--
--
-- Revisions:--
--
--
--
-- Additional Comments:
--
--
----------------------------------------------------------------------------------
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
use IEEE.numeric_std.all;
use IEEE.std_logic_arith.all;
entity RS232_Usr_Int is
            generic(
                                        Baud : integer := 9600;                                 --115,200, originally
9,600 bps
```

```
                                clk_in : integer := 25000000);                  --25MHz
    Port (          clk : in  STD_LOGIC;
                                rst : in  STD_LOGIC;
                                rs232_rcv : in  STD_LOGIC;
                                rs232_xmt : out  STD_LOGIC;
                                Data : INOUT  std_logic_vector(15 downto 0);
                                Addr : OUT  std_logic_vector(15 downto 0);
                                Xrqst : OUT  std_logic;
                                XDat : IN  std_logic;
                                YDat : OUT  std_logic;
                                BusRqst : OUT  std_logic;
                                BusCtrl : IN  std_logic
                        );
end RS232_Usr_Int;
architecture Behavioral of RS232_Usr_Int is
        type state_type is
(S0,S1,S2,S3,S4,S5,S6,S7,S8,S9,S10,S11,S12,S13,S14,S15,S16,S17,S18,S19,S20,S21,S22,S23,S24,
S25,S26,S27,S28,S29,S30,S31,S32,S33,S34,S35,S36,S37,S38,S39,
S40,S41,S42,S43,S44,S45,S46,S47,S48,S49,S50,S51,S52,S53,S54,S55,S56,S57,S58,S59,S60,S61,S6
2,S63,S64,S65,S67,S68,S69,  S70,S71,S72,S73,
S100,S101,S102,S103,S104,S105,S106,S107,S108,S109,S110,S111,S112,S113,S114,S115,S116,S117
,S118,S119,S120, S130, S131, S132, S133, S134, S135,S136,S137,S138,S139, S140, S141, S142,
S143, S144, S145, S146, S147, S148, S149, S150,
S300,S301,S302,S303,S304,S305,S306,S307,S308,S309,S310,S311,S312,S313,S314,S315,S316,S317
,S318,S319,S320,S321,S322,S323,S324,S325,S326,S327,S328,S329,S330,S331,S332,S333,S334,S33
5,S336,S337,S338,S339,S340,S400,S401,S402,S403,S404,S405,S406,S407,
E1,E2,E3,E4,E5,E6,E7,E8,E9,E10,E11,E12,E13,E14,E15,E16,E17,E18,E19,E20,E21,E22,E23,E24,E25,
D1,D2,D3,D4,D5,D6,D7,D8,D9,D10,D11,D12,D13,D14,D15,D16,D17,D18,D19,D20);
        signal CS_RS232_R, NS_RS232_R, CS_RS232_W, NS_RS232_W, CS_FIFO_Bus,
NS_FIFO_Bus: state_type;
        signal rx_done,tx_done :STD_LOGIC:= '0';
        signal temp_rcv: STD_LOGIC_VECTOR(7 downto 0):= (others => '0');
        signal i,j: STD_LOGIC_VECTOR (15 downto 0):= (others => '0');
        signal uartclk : STD_LOGIC:= '0';
        signal u: integer;
        signal rs232_rcv_s,rs232_rcv_t: STD_LOGIC:= '1';
        signal txbuff: STD_LOGIC_VECTOR(9 downto 0):= (others => '1');          --buff used to transmit 1
bytes with start and stop bits

        --Declare Signals for FIFO Serial Read
        signal STD_FIFO_R_WriteEn, STD_FIFO_R_ReadEn: STD_LOGIC:= '0';
        signal STD_FIFO_R_DataIn, STD_FIFO_R_DataOut: STD_LOGIC_VECTOR(7 downto 0):=
(others => '0');
        signal STD_FIFO_R_Empty, STD_FIFO_R_Full: STD_LOGIC:= '0';

        --Declare Signals for FIFO Serial Write
        signal STD_FIFO_W_WriteEn, STD_FIFO_W_ReadEn: STD_LOGIC:= '0';
        signal STD_FIFO_W_DataIn, STD_FIFO_W_DataOut: STD_LOGIC_VECTOR(7 downto 0):=
(others => '0');
        signal STD_FIFO_W_Empty, STD_FIFO_W_Full: STD_LOGIC:= '0';

        --Declare Signals for Bus Interface
        signal Bus_Int1_WE, Bus_Int1_RE, Bus_Int1_Busy: STD_LOGIC:= '0';
        signal Bus_Int1_DataIn, Bus_Int1_DataOut, Bus_Int1_AddrIn: STD_LOGIC_VECTOR(15 downto
0):= (others => '0');
```

```
        --Declare Signals for Registers
        signal LD_busy,LD_busy2,LD_rx,LD_tx,LD_temp_data,LD_temp2: STD_LOGIC:= '0';
        signal LD_Temp_Addr_High,LD_Temp_Addr_Low,LD_Temp_Data_High: STD_LOGIC:= '0';
        signal LD_Temp_Data_Low,ld_temp_cmd: STD_LOGIC:= '0';
        signal LD_Num_More,LD_Chk_Sum: STD_LOGIC:= '0';

        signal LD_Reg_Num_H,LD_Reg_Num_L: STD_LOGIC:= '0';
        signal LD_Reg_Num: STD_LOGIC:= '0';

        signal LD_Base_Addr: STD_LOGIC:= '0';
        signal LD_Reg_Addr_H,LD_Reg_Addr_L: STD_LOGIC:= '0';
        signal LD_Reg_Addr: STD_LOGIC:= '0';
        signal LD_Reg_Cnt: STD_LOGIC:= '0';
        signal LD_Data_Temp_H,LD_Data_Temp_L: STD_LOGIC:= '0';
        signal LD_CRC16_Temp: STD_LOGIC:= '0';
        signal LD_A_Poly: STD_LOGIC:= '0';
        signal LD_Is_Read: STD_LOGIC:= '0';
        signal LD_Carry_Over: STD_LOGIC:= '0';
        signal LD_CRC_Pass: STD_LOGIC:= '0';




        signal busy,busy_reg_o,busy2,busy2_reg_o,rx,rx_reg_o,tx,tx_reg_o: STD_LOGIC:= '0';
        signal temp_data_reg_o, temp_data: STD_LOGIC_VECTOR(15 downto 0):= (others => '0');
        signal temp2_reg_o, temp2: STD_LOGIC_VECTOR(7 downto 0):= (others => '0');
        signal Temp_Addr_High_reg_o, Temp_Addr_High: STD_LOGIC_VECTOR(7 downto 0):= (others
=> '0');
        signal Temp_Addr_Low_reg_o, Temp_Addr_Low: STD_LOGIC_VECTOR(7 downto 0):= (others
=> '0');
        signal Temp_Data_High_reg_o, Temp_Data_High: STD_LOGIC_VECTOR(7 downto 0):= (others
=> '0');
        signal Temp_Data_Low_reg_o, Temp_Data_Low: STD_LOGIC_VECTOR(7 downto 0):= (others
=> '0');
        signal Temp_Cmd_reg_o, Temp_Cmd: STD_LOGIC_VECTOR(7 downto 0):= (others => '0');

        signal Num_More_reg_o, Num_More: STD_LOGIC_VECTOR(7 downto 0):= (others => '0');

        signal Reg_Num_H_reg_o, Reg_Num_H:STD_LOGIC_VECTOR(7 downto 0):= (others => '0');
        signal Reg_Num_L_reg_o, Reg_Num_L:STD_LOGIC_VECTOR(7 downto 0):= (others => '0');

        signal Chk_Sum_reg_o, Chk_Sum: STD_LOGIC_VECTOR(15 downto 0):= (others => '0');
        signal Base_Addr_reg_o, Base_Addr: STD_LOGIC_VECTOR(15 downto 0):= (others => '0');
        signal Reg_Addr_H_reg_o, Reg_Addr_H: STD_LOGIC_VECTOR(7 downto 0):= (others => '0');
        signal Reg_Addr_L_reg_o, Reg_Addr_L: STD_LOGIC_VECTOR(7 downto 0):= (others => '0');
        signal Reg_Addr_reg_o, Reg_Addr: STD_LOGIC_VECTOR(15 downto 0):= (others => '0');
        signal Reg_Cnt_reg_o, Reg_Cnt: STD_LOGIC_VECTOR(7 downto 0):= (others => '0');
        signal Data_Temp_H_reg_o, Data_Temp_H: STD_LOGIC_VECTOR(7 downto 0):= (others =>
'0');
        signal Data_Temp_L_reg_o, Data_Temp_L: STD_LOGIC_VECTOR(7 downto 0):= (others =>
'0');


        signal CRC16_Temp_reg_o, CRC16_Temp: STD_LOGIC_VECTOR(15 downto 0):= (others =>
'0');
        signal A_Poly_reg_o, A_Poly: STD_LOGIC_VECTOR(15 downto 0):= (others => '0');
```

```
	signal Is_Read_reg_o, Is_Read: STD_LOGIC_VECTOR(7 downto 0):= (others => '0');
	signal Carry_Over_reg_o, Carry_Over: STD_LOGIC_VECTOR(7 downto 0):= (others => '0');

	signal CRC_Pass_reg_o, CRC_Pass: STD_LOGIC_VECTOR(7 downto 0):= (others => '0');

	--Signals for Counters
	signal Rcv_Cnt_rst,Rcv_Cnt_INC: STD_LOGIC:= '0';
	signal Rcv_Cnt_Out: STD_LOGIC_VECTOR(7 downto 0):= (others => '0');
	signal Buf_Cnt_rst,Buf_Cnt_INC: STD_LOGIC:= '0';
	signal Buf_Cnt_Out: STD_LOGIC_VECTOR(7 downto 0):= (others => '0');
	signal Reg_Cnt_rst,Reg_Cnt_INC: STD_LOGIC:= '0';
	signal Reg_Cnt_Out: STD_LOGIC_VECTOR(7 downto 0):= (others => '0');
	signal CRC_Cnt_rst,CRC_Cnt_INC: STD_LOGIC:= '0';
	signal CRC_Cnt_Out: STD_LOGIC_VECTOR(7 downto 0):= (others => '0');
	signal XOR_Cnt_rst,XOR_Cnt_INC: STD_LOGIC:= '0';
	signal XOR_Cnt_Out: STD_LOGIC_VECTOR(7 downto 0):= (others => '0');
	signal XOR2_Cnt_rst,XOR2_Cnt_INC: STD_LOGIC:= '0';
	signal XOR2_Cnt_Out: STD_LOGIC_VECTOR(7 downto 0):= (others => '0');
	signal RAM_Cnt_rst,RAM_Cnt_INC: STD_LOGIC:= '0';
	signal RAM_Cnt_Out: STD_LOGIC_VECTOR(7 downto 0):= (others => '0');


	--Signals for RAM (Storing Temp Packet Data) [256 Max]
	type ram_type is array (0 to (2**8)-1) of std_logic_vector(7 downto 0);
	signal RAM : ram_type;
	signal RAM_wea: STD_LOGIC:= '0';
	signal RAM_address,RAM_Data_In,RAM_Data_Out : std_logic_vector(7 downto 0);
	----User defined variables
	-- CM is the Clock Divder 25MHz/CM=115,200 Baud
	constant CM : integer := clk_in/Baud;
	-- CN is the read offset for serial input
	constant CN : integer :=CM/2;
	----End User defined variables


	--declare STD_FIFO
	COMPONENT STD_FIFO
	Generic (
		DATA_WIDTH		: integer;		-- Width of FIFO
		FIFO_DEPTH		: integer;		--	Depth of FIFO
		FIFO_ADDR_LEN  : integer		-- Required number of bits to represent
FIFO_Depth
	);
	Port (
		CLK	: in  STD_LOGIC;
		RST	: in  STD_LOGIC;
		WriteEn : in  STD_LOGIC;
		DataIn  : in  STD_LOGIC_VECTOR (7 downto 0);
		ReadEn  : in  STD_LOGIC;
		DataOut : out STD_LOGIC_VECTOR (7 downto 0);
		Empty   : out STD_LOGIC;
		Full    : out STD_LOGIC
	);
	end COMPONENT;

	--declare Bus Interface
```

```
        COMPONENT Bus_Int
  PORT(
                clk : IN  std_logic;
                rst : IN  std_logic;
                DataIn : IN  std_logic_vector(15 downto 0);
                DataOut : OUT  std_logic_vector(15 downto 0);
                AddrIn : IN  std_logic_vector(15 downto 0);
                WE : IN  std_logic;
                RE : IN  std_logic;
                Busy : OUT  std_logic;
                Data : INOUT  std_logic_vector(15 downto 0);
                Addr : OUT  std_logic_vector(15 downto 0);
                Xrqst : OUT  std_logic;
                XDat : IN  std_logic;
                YDat : OUT  std_logic;
                BusRqst : OUT  std_logic;
                BusCtrl : IN  std_logic
  );
   END COMPONENT;

        --declare Std_Counter Component
        component Std_Counter is
        generic
        (
                Width : integer              --width of counter
        );
        port(INC,rst,clk: in std_logic;
                 Count: out STD_LOGIC_VECTOR(Width-1 downto 0));
        end component;



begin
        --Instantiate STD_FIFO for Reading Serial Data
         STD_FIFO_R: STD_FIFO
         Generic Map
         (
                DATA_WIDTH   => 8,              -- Width of FIFO
                FIFO_DEPTH   => 512, --         Depth of FIFO
                FIFO_ADDR_LEN => 9 -- Required number of bits to represent FIFO_Depth
         )
         Port Map
         (
                CLK => clk,
                RST => rst,
                WriteEn => STD_FIFO_R_WriteEn,
                DataIn => STD_FIFO_R_DataIn,
                ReadEn => STD_FIFO_R_ReadEn,
                DataOut => STD_FIFO_R_DataOut,
                Empty   => STD_FIFO_R_Empty,
                Full   => STD_FIFO_R_Full
                );

        --Instantiate STD_FIFO for Writing Serial Data
         STD_FIFO_W: STD_FIFO
         Generic Map
```

```
    (
            DATA_WIDTH   => 8,              -- Width of FIFO
            FIFO_DEPTH   => 512, --         Depth of FIFO
            FIFO_ADDR_LEN => 9 -- Required number of bits to represent FIFO_Depth
    )
    Port Map(
            CLK => clk,
            RST => rst,
            WriteEn => STD_FIFO_W_WriteEn,
            DataIn => STD_FIFO_W_DataIn,
            ReadEn => STD_FIFO_W_ReadEn,
            DataOut => STD_FIFO_W_DataOut,
            Empty   => STD_FIFO_W_Empty,
            Full   => STD_FIFO_W_Full
            );
   --Instantiate Bus Interface
   Bus_Int1: Bus_Int PORT MAP (
  clk => clk,
  rst => rst,
  DataIn => Bus_Int1_DataIn,
  DataOut => Bus_Int1_DataOut,
  AddrIn => Bus_Int1_AddrIn,
  WE => Bus_Int1_WE,
  RE => Bus_Int1_RE,
  Busy => Bus_Int1_Busy,
  Data => Data,
  Addr => Addr,
  Xrqst => Xrqst,
  XDat => XDat,
  YDat => YDat,
  BusRqst => BusRqst,
  BusCtrl => BusCtrl
);

   --instantiate Rcv_Cnt_8
   Rcv_Cnt: Std_Counter
   generic map
   (
            Width => 8
   )
   port map(
                    clk => clk,
                    rst=> Rcv_Cnt_rst,
                    INC=> Rcv_Cnt_INC,
                    Count=>Rcv_Cnt_Out);

   --instantiate Buf_Cnt_8
   Buf_Cnt: Std_Counter
   generic map
   (
            Width => 8
   )
   port map(
                    clk => clk,
                    rst=> Buf_Cnt_rst,
                    INC=> Buf_Cnt_INC,
```

```
                Count=>Buf_Cnt_Out);

--instantiate Reg_Cnt_8
Reg_Cnt1: Std_Counter
generic map
(
        Width => 8
)
port map(
                clk => clk,
                rst=> Reg_Cnt_rst,
                INC=> Reg_Cnt_INC,
                Count=>Reg_Cnt_Out);

        --instantiate CRC_Cnt_8
CRC_Cnt: Std_Counter
generic map
(
        Width => 8
)
port map(
                clk => clk,
                rst=> CRC_Cnt_rst,
                INC=> CRC_Cnt_INC,
                Count=>CRC_Cnt_Out);

--instantiate XOR_Cnt_8
XOR_Cnt: Std_Counter
generic map
(
        Width => 8
)
port map(
                clk => clk,
                rst=> XOR_Cnt_rst,
                INC=> XOR_Cnt_INC,
                Count=>XOR_Cnt_Out);
--instantiate XOR_Cnt_8-2
XOR2_Cnt: Std_Counter
generic map
(
        Width => 8
)
port map(
                clk => clk,
                rst=> XOR2_Cnt_rst,
                INC=> XOR2_Cnt_INC,
                Count=>XOR2_Cnt_Out);
--instantiate RAM_Cnt_8-2
RAM_Cnt: Std_Counter
generic map
(
        Width => 8
)
port map(
                clk => clk,
```

```
                rst=> RAM_Cnt_rst,
                INC=> RAM_Cnt_INC,
                Count=>RAM_Cnt_Out);
----Registers
Reg_Proc: process
begin
        wait until clk'event and clk = '1';
        if rst = '0' then
                busy_reg_o <= '0';
                busy2_reg_o <='0';
                rx_reg_o <= '0';
                tx_reg_o <= '0';
                temp_data_reg_o <= (others => '0');
                temp2_reg_o <= (others => '0');
                Temp_Addr_High_reg_o <= (others => '0');
                Temp_Addr_Low_reg_o <= (others => '0');
                Temp_Data_High_reg_o <= (others => '0');
                Temp_Data_Low_reg_o <= (others => '0');
                Temp_Cmd_reg_o <= (others => '0');
                Num_More_reg_o <= (others => '0');

                Reg_Num_H_reg_o <= (others => '0');
                Reg_Num_L_reg_o <= (others => '0');

                Chk_Sum_reg_o <= (others => '0');
                Reg_Addr_H_reg_o <= (others => '0');
                Reg_Addr_L_reg_o <= (others => '0');
                Reg_Addr_reg_o <= (others => '0');
                Data_Temp_H_reg_o <= (others => '0');
                Data_Temp_L_reg_o <= (others => '0');
                Reg_Cnt_reg_o <= (others => '0');
                Base_Addr_reg_o <= (others => '0');

                CRC16_Temp_reg_o <= (others => '0');
                A_Poly_reg_o <= (others => '0');
                Is_Read_reg_o <= (others => '0');
                Carry_Over_reg_o <= (others => '0');
                CRC_Pass_reg_o <= (others => '0');



        else
        if (LD_busy = '1') then busy_reg_o <= busy; end if;
        if (LD_busy2 = '1') then busy2_reg_o <= busy2; end if;
        if (LD_rx = '1') then rx_reg_o <= rx; end if;
        if (LD_tx = '1') then tx_reg_o <= tx; end if;
        if (LD_temp_data = '1') then temp_data_reg_o <= temp_data; end if;
        if (LD_temp2 = '1') then temp2_reg_o <= temp2; end if;
        if (LD_Temp_Addr_High = '1') then Temp_Addr_High_reg_o <= Temp_Addr_High; end if;
        if (LD_Temp_Addr_Low = '1') then Temp_Addr_Low_reg_o <= Temp_Addr_Low; end if;
        if (LD_Temp_Data_High = '1') then Temp_Data_High_reg_o <= Temp_Data_High; end if;
        if (LD_Temp_Data_Low = '1') then Temp_Data_Low_reg_o <= Temp_Data_Low; end if;
        if (LD_Temp_Cmd = '1') then Temp_Cmd_reg_o <= Temp_Cmd; end if;
        if (LD_Num_More = '1') then Num_More_reg_o <= Num_More; end if;
```

```
		if (LD_Reg_Num_H = '1') then Reg_Num_H_reg_o <= Reg_Num_H; end if;
		if (LD_Reg_Num_L = '1') then Reg_Num_L_reg_o <= Reg_Num_L; end if;

		if (LD_Chk_Sum = '1') then Chk_Sum_reg_o <= Chk_Sum; end if;
		if (LD_Reg_Addr_H = '1') then Reg_Addr_H_reg_o <= Reg_Addr_H; end if;
		if (LD_Reg_Addr_L = '1') then Reg_Addr_L_reg_o <= Reg_Addr_L; end if;
		if (LD_Reg_Addr = '1') then Reg_Addr_reg_o <= Reg_Addr; end if;
		if (LD_Data_Temp_H = '1') then Data_Temp_H_reg_o <= Data_Temp_H; end if;
		if (LD_Data_Temp_L = '1') then Data_Temp_L_reg_o <= Data_Temp_L; end if;
		if (LD_Reg_Cnt = '1') then Reg_Cnt_reg_o <= Reg_Cnt; end if;
		if (LD_Base_Addr = '1') then Base_Addr_reg_o <= Base_Addr; end if;
		if (LD_CRC16_Temp = '1') then CRC16_Temp_reg_o <= CRC16_Temp; end if;
		if (LD_A_Poly = '1') then A_Poly_reg_o <= A_Poly; end if;

		if (LD_Is_Read = '1') then Is_Read_reg_o <= Is_Read; end if;
		if (LD_Carry_Over = '1') then Carry_Over_reg_o <= Carry_Over; end if;
		if (LD_CRC_Pass = '1') then CRC_Pass_reg_o <= CRC_Pass; end if;


	end if;
	end process;
	----End Registers

	----Process Memory Data_Reg: process
	Ram_Proc: process
	begin
		wait until clk'event and clk = '1';

		if (RAM_wea = '1') then
			RAM(conv_integer(RAM_address)) <= RAM_Data_In;
		end if;
		RAM_Data_Out <= RAM(conv_integer(RAM_address));
	end process Ram_Proc;
	----End Ram

	----Next State Logic for Serial Interface Read
	NSL_RS232_R: process(CS_RS232_R,rs232_rcv_s,rx_done,STD_FIFO_R_Full,temp_rcv)
	begin


		----Default States to remove latches
		busy <='0';
		rx <= '0';
		NS_RS232_R <= S0;
		LD_busy <= '0';
		LD_rx <= '0';

		--Signals for FIFO
		STD_FIFO_R_WriteEn <='0';
		STD_FIFO_R_DataIn<= (others => '0');

		case CS_RS232_R is
			when S0 =>                                                    -- Waits until
data is detected on rs232_rcv_s.
				if (rs232_rcv_s = '1') then
```

```
                                        NS_RS232_R <= S0;
                                else
                                        NS_RS232_R <= S1;
                                end if;
                                        busy <='0';                                     -- the busy
signal stops the baud generator
                                        rx <= '0';                                      -- signals to
stop reading data
                                        LD_rx <= '1';
                                        LD_busy <= '1';
                        when S1=>                                                       -- Starts the
baud rate generator and reading
                                        NS_RS232_R<=S2;
                                        busy <='1';                                     -- the busy
signal starts the baud generator
                                        rx <= '1';                                      -- signals to
start reading data
                                        LD_rx <= '1';
                                        LD_busy <= '1';

                        when S2 =>                                                      -- Waits until all
data is read
                                        if (rx_done ='0') then
                                                NS_RS232_R<= S2;
                                        else
                                                NS_RS232_R<=S3;
                                        end if;

                        when S3 =>
                                        if (STD_FIFO_R_Full = '0') then
                                                STD_FIFO_R_DataIn <= temp_rcv;
                                                STD_FIFO_R_WriteEn <='1';
                                        end if;
                                        NS_RS232_R<=S0;
                        when others =>
                                NS_RS232_R<=S0;

                end case;
        end process;
        ----End Next State Logic for Serial Interface Read
        ----Next State Logic for Serial Interface Write
        NSL_RS232_W: process(CS_RS232_W,tx_done,STD_FIFO_W_Empty,STD_FIFO_W_DataOut)
        begin

                ----Default States to remove latches
                tx<='0';
                NS_RS232_W <= S0;
                temp2 <= (others => '0');
                LD_tx <= '0';
                LD_temp2<='0';
                Busy2 <='0';
                LD_Busy2<='0';

                --Signals for FIFO
                STD_FIFO_W_ReadEn <= '0';
                case CS_RS232_W is
```

```
                        when S0=>
                                if(STD_FIFO_W_Empty = '1') then
                                        NS_RS232_W<=S0;
                                else
                                        NS_RS232_W<=S1;
                                        STD_FIFO_W_ReadEn <= '1';
                                end if;
                                busy2 <='0';                                    -- the busy signal stops
the baud generator
                                tx <= '0';                                              -- signals to
stop sending data
                                LD_tx <= '1';
                                LD_busy2 <= '1';

                        when S1=>
                                temp2<=STD_FIFO_W_DataOut;
                                LD_temp2<='1';
                                NS_RS232_W<=S2;

                        when S2=>
                                busy2 <='1';                                    -- the busy signal starts
the baud generator
                                tx<='1';                                                -- signals to
start sending data
                                LD_tx<='1';
                                LD_busy2 <= '1';
                                NS_RS232_W<=S3;

                        when S3=>
                                if(tx_done='0') then
                                        NS_RS232_W <=S3;
                                else
                                        NS_RS232_W <=S0;
                                end if;
                        when others =>
                                NS_RS232_W <=S0;
                end case;
        end process;
        ----End Next State Logic for Serial Interface Write
        ----Next State Logic for FIFO to Bus
        NSL_FIFO_Bus: process(CS_FIFO_Bus,
STD_FIFO_R_Empty,Temp_Cmd_reg_o,Bus_Int1_Busy,STD_FIFO_R_DataOut,Temp_Addr_High_reg_
o,Temp_Addr_Low_reg_o,Temp_Data_High_reg_o,Temp_Data_Low_reg_o,Temp_Data_reg_o,Bus_Int1
_DataOut,temp_data_reg_o,Num_More_reg_o,Chk_Sum_reg_o,Rcv_Cnt_Out,RAM_Data_Out,Reg_Cnt
_reg_o,Reg_Addr_H_reg_o,Reg_Addr_L_reg_o,Base_Addr_reg_o, Reg_Num_H_reg_o,
Reg_Num_L_reg_o )
        begin
                ----Default States to remove latches
                NS_FIFO_Bus <=S0;
                Temp_Cmd <= (others => '0');
                LD_Temp_Cmd <='0';
                Temp_Addr_High <= (others => '0');
                LD_Temp_Addr_High <='0';
                Temp_Addr_Low <= (others => '0');
                LD_Temp_Addr_Low <='0';
```

```
Bus_Int1_AddrIn <= (others => '0');
Bus_Int1_RE <='0';
Bus_Int1_DataIn <= (others => '0');
Bus_Int1_WE <='0';
Temp_Data <= (others => '0');
LD_Temp_Data <='0';
Temp_Data_High <= (others => '0');
LD_Temp_Data_High <='0';
Temp_Data_Low<= (others => '0');
LD_Temp_Data_Low <='0';

--Signals for FIFO
STD_FIFO_R_ReadEn <='0';
STD_FIFO_W_DataIn <= (others => '0');
STD_FIFO_W_WriteEn <='0';

--Signals for Counters
Rcv_Cnt_rst<='1';
Rcv_Cnt_INC<='0';
Buf_Cnt_rst<='1';
Buf_Cnt_INC<='0';
Reg_Cnt_rst<='1';
Reg_Cnt_INC<='0';
CRC_Cnt_rst<='1';
CRC_Cnt_INC<='0';
XOR_Cnt_rst<='1';
XOR_Cnt_INC<='0';
XOR2_Cnt_rst<='1';
XOR2_Cnt_INC<='0';
RAM_Cnt_rst<='1';
RAM_Cnt_INC<='0';


--Signals for memory
RAM_address <= (others => '0');
RAM_Data_In <= (others => '0');
RAM_wea <= '0';

Num_More <= (others => '0');
LD_Num_More <='0';
Chk_Sum <= (others => '0');
LD_Chk_Sum <='0';


Reg_Num_H <= (others => '0');
LD_Reg_Num_H <='0';
Reg_Num_L <= (others => '0');
LD_Reg_Num_L <='0';

Reg_Addr_H <= (others => '0');
LD_Reg_Addr_H <='0';
Reg_Addr_L <= (others => '0');
LD_Reg_Addr_L <='0';
Reg_Addr <= (others => '0');
LD_Reg_Addr <='0';
Data_Temp_H <= (others => '0');
```

```
LD_Data_Temp_H <='0';
Data_Temp_L <= (others => '0');
LD_Data_Temp_L <='0';
Reg_Cnt <= (others => '0');
LD_Reg_Cnt <='0';
Base_Addr <= (others => '0');
LD_Base_Addr <='0';
CRC16_Temp <= (others => '0');
LD_CRC16_Temp <= '0';
A_Poly <= (others => '0');
LD_A_Poly <= '0';



Is_Read <= (others => '0');
Carry_Over <= (others => '0');
CRC_Pass <= (others => '0');

LD_Is_Read <= '0';
LD_Carry_Over <= '0';
LD_CRC_Pass <= '0';


case CS_FIFO_Bus is

        when S0=>
                if(STD_FIFO_R_Empty = '1') then          --Check to see if
commands are in queue
                        NS_FIFO_Bus<=S0;
                else
                        NS_FIFO_Bus<=S1;
                        STD_FIFO_R_ReadEn <= '1';        --Assert Read Signal for
FIFO
                end if;
                Rcv_Cnt_rst<='0';
                Buf_Cnt_rst<='0';
                Reg_Cnt_rst<='0';
                RAM_Cnt_rst<='0';

        when S1=>                                                --Read
Command from FIFO
                Temp_Cmd<=STD_FIFO_R_DataOut;            -- Storing
Device Address
                LD_Temp_Cmd<='1';
                RAM_Data_In <= STD_FIFO_R_DataOut;               --
Device Address, Ram Address 0X0000
                RAM_address <= X"00";
                RAM_wea <='1';

                NS_FIFO_Bus<=S2;


        when S2=>                                                --
Empty case state to allow registers to stabalize
                NS_FIFO_Bus <= S3;
```

```
                when S3=>
                        if(STD_FIFO_R_Empty = '1') then         --Check to see if
commands are in queue
                                NS_FIFO_Bus<=S3;
                        else
                                NS_FIFO_Bus<=S4;
                                STD_FIFO_R_ReadEn <= '1';       --Assert Read Signal for
FIFO
                        end if;


                when S4=>                                                       --Read
Command from FIFO
                        RAM_Data_In <= STD_FIFO_R_DataOut;                      --
Functional Code, Ram Address 0X0001
                        RAM_address <= X"01";
                        RAM_wea <='1';
                        Temp_Cmd <= STD_FIFO_R_DataOut;
                        LD_Temp_Cmd <= '1';

                        NS_FIFO_Bus<=S40;


                when S40=>
                        if(STD_FIFO_R_Empty = '1') then         --Check to see if
commands are in queue
                                NS_FIFO_Bus<=S40;
                        else
                                NS_FIFO_Bus<=S41;
                                STD_FIFO_R_ReadEn <= '1';               --Assert Read
Signal for FIFO
                        end if;


                when S41=>                                                      --Load
Starting Address High
                        Reg_Addr_H <= STD_FIFO_R_DataOut;
                        LD_Reg_Addr_H <='1';
                        Temp_Cmd <= STD_FIFO_R_DataOut;
                        LD_Temp_Cmd <= '1';
                        RAM_Data_In <= STD_FIFO_R_DataOut;                      --
Starting Address High, Ram Address 0X0002
                        RAM_address <= X"02";
                        RAM_wea <='1';
                        NS_FIFO_Bus <= S42;


                when S42=>
                        if(STD_FIFO_R_Empty = '1') then         --Check to see if
commands are in queue
                                NS_FIFO_Bus<=S42;
                        else
                                NS_FIFO_Bus<=S43;
                                STD_FIFO_R_ReadEn <= '1';               --Assert Read
Signal for FIFO
```

```
                end if;


            when S43=>                                                  --Load
Starting Address Low
                Reg_Addr_L <= STD_FIFO_R_DataOut;
                LD_Reg_Addr_L <='1';
                Temp_Cmd <= STD_FIFO_R_DataOut;
                LD_Temp_Cmd <= '1';
                RAM_Data_In <= STD_FIFO_R_DataOut;                      --
Starting Address Low, Ram Address 0X0003
                RAM_address <= X"03";
                RAM_wea <='1';

                NS_FIFO_Bus <= S44;

            when S44=>
                if(STD_FIFO_R_Empty = '1') then             --Check to see if
commands are in queue
                    NS_FIFO_Bus<=S44;
                else
                    NS_FIFO_Bus<=S45;
                    STD_FIFO_R_ReadEn <= '1';                   --Assert Read
Signal for FIFO
                end if;

            when S45=>
                Base_Addr(15 downto 8) <= Reg_Addr_H_reg_o;
                Base_Addr(7 downto 0) <= Reg_Addr_L_reg_o;
                LD_Base_Addr <= '1';
                NS_FIFO_Bus <= S46;


            -- Here add and store the Num of Reg Hi and Lo This is used in the write
response
            when S46=>                                                  --Load
Number of Registers High
                Reg_Num_H <= STD_FIFO_R_DataOut;
                LD_Reg_Num_H <='1';
                Temp_Cmd <= STD_FIFO_R_DataOut;
                LD_Temp_Cmd <= '1';
                RAM_Data_In <= STD_FIFO_R_DataOut;                      --
Number of Registers High, Ram Address 0X0004
                RAM_address <= X"04";
                RAM_wea <='1';
                NS_FIFO_Bus <= S47;


            when S47=>
                if(STD_FIFO_R_Empty = '1') then             --Check to see if
commands are in queue
                    NS_FIFO_Bus<=S47;
                else
                    NS_FIFO_Bus<=S48;
                    STD_FIFO_R_ReadEn <= '1';                   --Assert Read
Signal for FIFO
```

```
                        end if;


                when S48=>                                                      --Load
Number of Registers Low
                        Reg_Num_L <= STD_FIFO_R_DataOut;

                        Reg_Cnt <= Reg_Num_L + Reg_Num_L;       -- Register Count =
Number of Bytes more = 2 * Register Number Low

                        LD_Reg_Num_L <='1';
                        LD_Reg_Cnt<='1';

                        Temp_Cmd <= STD_FIFO_R_DataOut;
                        LD_Temp_Cmd <= '1';

                        RAM_Data_In <= STD_FIFO_R_DataOut;                      --
Number of Registers Low, Ram Address 0X0005
                        RAM_address <= X"05";
                        RAM_wea <='1';

                        NS_FIFO_Bus <= S49;



                when S49=>
                        if(STD_FIFO_R_Empty = '1') then         --Check to see if
commands are in queue
                                NS_FIFO_Bus<=S49;
                        else
                                NS_FIFO_Bus<=S50;
                                STD_FIFO_R_ReadEn <= '1';               --Assert Read
Signal for FIFO
                        end if;



                when S50=>
                        RAM_address <= X"01";                           -- Set
RAM_address to function code for routing
                        NS_FIFO_Bus<=S51;


                -- If this is a write request, store the value of "Number of Bytes further" into
Num_More

                when S51=>
                        if(RAM_Data_Out = X"03") then           --Check Cmd (Read) - Original
UART is 0x0F for Read
                                Is_Read <= X"01";
                                LD_Is_Read <= '1';
                                NS_FIFO_Bus <= S60;
                        elsif(RAM_Data_Out = X"10") then        --Check Cmd (Write) - Original
UART is 0x0A for Write
                                Is_Read <= X"00";
                                LD_Is_Read <= '1';
```

```
                                NS_FIFO_Bus <= S52;
                        else                                                    --Check
Cmd (Invalid Data)
                                NS_FIFO_Bus <= S0;
                        end if;


                when S52=>
                        RAM_address <= X"06";                   -- Set RAM_address to
06

                        NS_FIFO_Bus <= S53;

                when S53=>
                        Num_More<=STD_FIFO_R_DataOut;           -- Num_More =
Num Bytes MORE
                        Reg_Cnt<=Num_More;                              --
Setting Reg Cnt to Pkt Len for testing
                        LD_Num_More<='1';
                        LD_Reg_Cnt<='1';
                        Temp_Cmd <= STD_FIFO_R_DataOut;
                        LD_Temp_Cmd <= '1';
                        RAM_Data_In <= STD_FIFO_R_DataOut;              -- Num
More Bytes (Num_More), Ram Address 0X0006   NOTE, Num More Byes 0X06 is for WRITE ONLY
                        RAM_address <= X"06";
                        RAM_wea <='1';

                        NS_FIFO_Bus<=S54;

                when S54=>
                        RAM_address <= X"01";                   -- Set Ram
Address to 0X01 for Function Code for state routing

                        NS_FIFO_Bus<=S5;



                when S60=>
                        Chk_Sum <= X"0000";
                        LD_Chk_Sum <= '1';
                        NS_FIFO_Bus <= S61;

                when S61=>
                        NS_FIFO_Bus <= S62;

                when S62=>
                        Chk_Sum <= STD_FIFO_R_DataOut & Chk_Sum_reg_o(15 downto 8);
        --High Byte of CRC added to Low Byte of Chk_Sum_reg_o
                        Temp_Cmd <= STD_FIFO_R_DataOut;
                        LD_Temp_Cmd <= '1';
                        LD_Chk_Sum <= '1';
                        NS_FIFO_Bus <= S63;

                when S63=>
                        if(STD_FIFO_R_Empty = '1') then         --Check to see if
commands are in queue
```

```
                                        NS_FIFO_Bus<=S63;
                                    else
                                        NS_FIFO_Bus<=S64;
                                        STD_FIFO_R_ReadEn <= '1';                   --Assert Read
Signal for FIFO
                                    end if;

                                when S64=>
                                    Chk_Sum <= STD_FIFO_R_DataOut & Chk_Sum_reg_o(15 downto 8);
        --Low Byte of CRC added to High Byte of Chk_Sum_reg_o
                                    LD_Chk_Sum <= '1';
                                    Temp_Cmd <= STD_FIFO_R_DataOut;
                                    LD_Temp_Cmd <= '1';
                                    NS_FIFO_Bus <= S300;



                                ----------------------------



                                when S5=>
                                    RAM_address <= X"01";
                                    if((Num_More_reg_o < X"FF") and (Num_More_reg_o > X"03")) then
        --Check Packet Length (Packet Length < 255 bytes) [0xFE is Max; 0x04 is Min]
                                        NS_FIFO_Bus <= S6;
                                        -- Code that skips from S5 to S300 for read requests
                                        if(RAM_Data_Out = X"03") then
                                            NS_FIFO_Bus <= S300;
                                        end if;
                                    else                                                    --Check
Cmd (Invalid Data)
                                        NS_FIFO_Bus <= S0;
                                    end if;
                                    Chk_Sum <= X"0000";
                                    LD_Chk_Sum <='1';
                                    Rcv_Cnt_rst<='0';                                   --Active Low

                                    RAM_address <= X"01";


                                -- S6-S9: Process to collect incoming data
                                when S6=>
                                    if(Rcv_Cnt_Out < (Num_More_reg_o+2)) then            --Read data to
memory
                                        NS_FIFO_Bus <= S7;
                                    else
                                        NS_FIFO_Bus <= S300;
                                    end if;

                                when S7=>
                                    if(STD_FIFO_R_Empty = '1') then                 --Check to see if
commands are in queue
                                        NS_FIFO_Bus<=S7;
                                    else
                                        NS_FIFO_Bus<=S8;
```

```
                                        STD_FIFO_R_ReadEn <= '1';                       --Assert Read
Signal for FIFO
                                end if;

                        when S8=>
        --Read Data from FIFO to memory and calc checksum
                                if(Rcv_Cnt_Out > (Num_More_reg_o - 1)) then -- If we are past the data,
load in check sum
                                        Chk_Sum <= STD_FIFO_R_DataOut & Chk_Sum_reg_o(15
downto 8);
                                        LD_Chk_Sum <= '1';
                                end if;
                                RAM_Data_In <= STD_FIFO_R_DataOut;
                                -- Here the RAM address offset will be + 7, RAM_address 0X00 to 0X06
consist packet information (not data)
                                RAM_address <= Rcv_Cnt_Out + 7;                                 --
Changed from 1 to 7, storing all pack information in RAM
                                RAM_wea <='1';
                                Temp_Cmd <= STD_FIFO_R_DataOut;
                                LD_Temp_Cmd <= '1';
                                Rcv_Cnt_INC<='1';
                                NS_FIFO_Bus <= S6;

        ---------- Adjusting Chk_sum. Forcing a pass of S9


                        ---------------------------------------------------------------------------
                        -- CRC16 Forming for Commands. This is used to check the calculated CRC16
(CRC16_Temp_reg_o) vs the....
                        --      given CRC16 value (Chk_Sum_reg_o)
                        ---------------------------------------------------------------------------

                        when S300=>                                                     -- Reset
CRC16_Temp_reg_o and Set A_Poly_reg_o
                                --Statements for Testing
                                --STD_FIFO_W_DataIn<= X"AA";
                                --STD_FIFO_W_WriteEn <='1';


                                CRC16_Temp <= X"FFFF";                                  -- Load
CRC16_Temp_reg_o with all 1's
                                LD_CRC16_Temp <= '1';
                                A_Poly <= X"A001";                                      -- Load A_Poly
                                LD_A_Poly <= '1';
                                CRC_Cnt_rst <= '0';                                     -- Active Low
                                XOR_Cnt_rst <= '0';
                                XOR2_Cnt_rst <= '0';
                                CRC_Pass <= X"00";
                                LD_CRC_Pass <= '1';
                                Ram_address <= X"00";                           -- Reset RAM Address
to 0. First data point at address 1, next state increments to 1
                                NS_FIFO_Bus <= S301;

                        when S301=>                                                     -- Check if there
is more data to process
                                Ram_address <= XOR_Cnt_Out;
```

```
                        NS_FIFO_Bus <= S335;

                when S335 =>
                        Ram_address <= XOR_Cnt_Out;
                        NS_FIFO_Bus <= S302;

                when S302=>                                             -- XOR
CRC16_Temp with Data Register (16 bit XOR 8 bit), XOR's one data register with CRC16_Temp
                        --Statements for Testing
                        --STD_FIFO_W_DataIn<= X"BB";
                        --STD_FIFO_W_WriteEn <='1';


                        CRC16_Temp <=  CRC16_Temp_reg_o XOR (X"00" & RAM_Data_Out);
-- LEFT OF HERE
                        LD_CRC16_Temp <= '1';
                        NS_FIFO_Bus <= S331;

                when S331=>
                        if(CRC16_Temp_reg_o(0) = '1') then
                                Carry_Over <= X"01";
                                LD_Carry_Over <= '1';
                        end if;
                        NS_FIFO_Bus <= S303;


                when S303=>                                             -- Shift
CRC16_Temp_reg_o to the right
                        CRC16_Temp <= '0' & CRC16_Temp_reg_o(15 downto 1);
                        LD_CRC16_Temp <= '1';
                        NS_FIFO_Bus <= S304;

                when S304 =>                                            -- Check for Carry Over
in CRC16_Temp_reg_o
                        --Statements for Testing
                        --STD_FIFO_W_DataIn<= X"C0";
                        --STD_FIFO_W_WriteEn <='1';

                        if(Carry_Over_reg_o = X"01") then
                                NS_FIFO_Bus <= S305;
                        else
                                NS_FIFO_Bus <= S306;
                        end if;

                when S305 =>                                            -- Carry Over was
positive, CRC16 XOR POLY
                                CRC16_Temp <= CRC16_Temp_reg_o xor A_Poly_reg_o;
                                LD_CRC16_Temp <= '1';
                                Carry_Over <= X"00";
                                LD_Carry_Over <= '1';
                                NS_FIFO_Bus <= S306;

                when S306 =>                                            -- Increment CRC_Cnt
                        CRC_Cnt_INC <= '1';
                        NS_FIFO_Bus <= S320;
```

```
when S320 =>
        NS_FIFO_Bus <= S307;

when S307 =>                                            -- Check if CRC_Cnt >
7
        if(CRC_Cnt_Out > 7) then
                NS_FIFO_Bus <= S308;
        else
                NS_FIFO_Bus <= S331;
        end if;
        Ram_address <= XOR_Cnt_Out;

when S308 =>
        if(Is_Read_reg_o = X"01") then          -- Read
                NS_FIFO_Bus <= S309;
        elsif(Is_Read_reg_o = X"00") then       -- Write
                NS_FIFO_Bus <= S333;
        else
                NS_FIFO_Bus <= S0;              -- Somethign went wrong, reset
        end if;
        CRC_Cnt_rst <= '0';
        Ram_address <= XOR_Cnt_Out;




when S309 =>
        if(XOR_Cnt_Out = X"05") then
                NS_FIFO_Bus <= D6;    -- End of write message
        else
                XOR_Cnt_INC <= '1';
                NS_FIFO_Bus <= S310;
        end if;

when S310 =>
        Ram_Address <= XOR_Cnt_Out;
        NS_FIFO_Bus <= S330;

when S330 =>
        Ram_Address <= XOR_Cnt_Out;
        NS_FIFO_Bus <= S302;

-- write
when S333 =>
        Ram_Address <= XOR_Cnt_Out;
        NS_FIFO_Bus <= S400;

when S400 =>
        --Statements for Testing
        --STD_FIFO_W_DataIn<= XOR_Cnt_Out;
        --STD_FIFO_W_WriteEn <='1';


        Ram_Address <= XOR_Cnt_Out;
        NS_FIFO_Bus <= S401;
```

```
                when S401 =>
                        --Statements for Testing
                        --STD_FIFO_W_DataIn<= Ram_Address;
                        --STD_FIFO_W_WriteEn <='1';

                        --Ram_Address <= XOR_Cnt_Out;
                        NS_FIFO_Bus <= S311;


                when S311 =>
                        --Statements for Testing
                        --STD_FIFO_W_DataIn<= XOR_Cnt_Out;
                        --STD_FIFO_W_WriteEn <='1';


                        if(XOR_Cnt_Out < (Reg_Num_L_reg_o + Reg_Num_L_reg_o + 6)) then
                        --Statements for Testing
                                --STD_FIFO_W_DataIn<= RAM_Address;
                                --STD_FIFO_W_WriteEn <='1';

                                XOR_Cnt_INC <= '1';
                                NS_FIFO_Bus <= S336;
                        else
                                NS_FIFO_Bus <= D6;
                                --NS_FIFO_Bus <= S9;
                        end if;


                when S336 =>
                        --Statements for Testing
                        --STD_FIFO_W_DataIn<= X"DD";
                        --STD_FIFO_W_WriteEn <='1';

                        NS_FIFO_Bus <= S301;

                -- Sequence for 3.5 Char Delay before response message sent
                when D6 =>
                        XOR2_Cnt_rst<='0';
                        NS_FIFO_Bus <= D7;

                when D7 =>
                        if(XOR2_Cnt_Out < 52500) then           -- 52500 = 3.675 Char @ 9600
Baud, minimum of 3.5 Char
                                XOR2_Cnt_INC <= '1';
                                NS_FIFO_Bus<= D8;
                        else
                                NS_FIFO_Bus <= S312;
                                XOR2_Cnt_rst<='0';
                        end if;

                When D8 =>
                        NS_FIFO_Bus <= D7;
```

```
                when S312 =>                                    -- Check if Received
CRC is correct
                        --Statements for Testing
                        --STD_FIFO_W_DataIn<= X"CC";
                        --STD_FIFO_W_WriteEn <='1';

                        if(CRC16_Temp_reg_o = Chk_Sum_reg_o) then
                                CRC_Pass <= X"01";                              --
CRC16 Correct
                                LD_CRC_Pass <= '1';
                                NS_FIFO_Bus <= S10;
                        else
                                CRC_Pass <= X"00";                              --
CRC16 WRONG
                                LD_CRC_Pass <= '1';
                                NS_FIFO_Bus <= E1;
                        end if;
                        Rcv_Cnt_rst<='0';
                        RAM_address <= X"01";


                ----------------------------------------------------------------------------------------------------

                --Error response starts on S313
                -- Write Response after saving data

                when E1 =>                                      -- Set Ram
Address to 0X00, Device ID
                        RAM_address <= X"00";
                        NS_FIFO_Bus<=E2;

                when E2=>                                       --Send
First byte of Packet(Device ID)
                        STD_FIFO_W_DataIn<= RAM_Data_Out;
                        STD_FIFO_W_WriteEn <='1';
                        NS_FIFO_Bus<=E3;

                when E3=>
                        RAM_address <= X"01";
                        NS_FIFO_Bus<= E4;

                when E4=>
                        RAM_Data_In <= RAM_Data_Out + 128;      -- OP ID, error changes
high-order bit to 1
                        RAM_address <= X"01";
                        RAM_wea <='1';
                        NS_FIFO_Bus<=E5;

                when E5=>                                       --
Timing delay
                        RAM_address <= X"01";
                        NS_FIFO_Bus<=E6;


                when E6=>                                       --Send
Second byte of Packet (OP_ID), error changes high-order bit to 1
```

```
                        STD_FIFO_W_DataIn <= RAM_Data_Out;        --Send OP_ID, error
changes high-order bit to 1
                        STD_FIFO_W_WriteEn <='1';
                        NS_FIFO_Bus<=E7;

                when E7=>                                                   --Send
Fourth byte of Packet - Error Code
                        STD_FIFO_W_DataIn <= X"01";--Send Error Code, Currently defaulting
to 01 = "FUNCTION CODE ACCEPTED CAN NOT BE PROCESSED"
                        STD_FIFO_W_WriteEn <='1';
                        NS_FIFO_Bus<=E8;

                -- Calculate CRC16 for Error Response (Starts at state E8)
                when E8=>                                                   -- Reset
CRC16_Temp_reg_o and Set A_Poly_reg_o
                        CRC16_Temp <= X"FFFF";                              -- Load
CRC16_Temp_reg_o with all 1's
                        LD_CRC16_Temp <= '1';
                        A_Poly <= X"A001";                                  -- Load A_Poly
                        LD_A_Poly <= '1';
                        CRC_Cnt_rst <= '0';                                 -- Active Low
                        XOR_Cnt_rst <= '0';
                        XOR2_Cnt_rst <= '0';
                        Ram_address <= X"00";                       -- Reset RAM Address
to 0. First data point at address 1, next state increments to 1
                        NS_FIFO_Bus <= E9;

                when E9=>                                                   -- Check if there
is more data to process
                        Ram_address <= XOR_Cnt_Out;
                        NS_FIFO_Bus <= E10;

                when E10 =>
                        Ram_address <= XOR_Cnt_Out;
                        NS_FIFO_Bus <= E11;

                when E11=>                                                  -- XOR
CRC16_Temp with Data Register (16 bit XOR 8 bit), XOR's one data register with CRC16_Temp
                                CRC16_Temp <=  CRC16_Temp_reg_o XOR (X"00" &
RAM_Data_Out);
                                LD_CRC16_Temp <= '1';
                                NS_FIFO_Bus <= E12;

                when E12=>
                        if(CRC16_Temp_reg_o(0) = '1') then
                                Carry_Over <= X"01";
                                LD_Carry_Over <= '1';
                        end if;
                        NS_FIFO_Bus <= E13;


                when E13=>                                                  -- Shift
CRC16_Temp_reg_o to the right
                        CRC16_Temp <= '0' & CRC16_Temp_reg_o(15 downto 1);
                        LD_CRC16_Temp <= '1';
                        NS_FIFO_Bus <= E14;
```

```
                when E14 =>                                             -- Check for
Carry Over in CRC16_Temp_reg_o
                        if(Carry_Over_reg_o = X"01") then
                                NS_FIFO_Bus <= E15;
                        else
                                NS_FIFO_Bus <= E16;
                        end if;

                when E15 =>                                             -- Carry Over was
positive, CRC16 XOR POLY
                                CRC16_Temp <= CRC16_Temp_reg_o xor A_Poly_reg_o;
                                LD_CRC16_Temp <= '1';
                                Carry_Over <= X"00";
                                LD_Carry_Over <= '1';
                                NS_FIFO_Bus <= E16;

                when E16 =>                                             -- Increment CRC_Cnt
                        CRC_Cnt_INC <= '1';
                        NS_FIFO_Bus <= E17;

                when E17 =>
                        NS_FIFO_Bus <= E18;

                when E18 =>                                             -- Check if CRC_Cnt >
7
                        if(CRC_Cnt_Out > 7) then
                                NS_FIFO_Bus <= E19;
                        else
                                NS_FIFO_Bus <= E12;
                        end if;
                        Ram_address <= XOR_Cnt_Out;


                when E19 =>
                        CRC_Cnt_rst <= '0';
                        Ram_address <= XOR_Cnt_Out;
                        NS_FIFO_Bus <= E20;


                when E20 =>
                        if(XOR_Cnt_Out < 2) then
                                XOR_Cnt_INC <= '1';
                                NS_FIFO_Bus <= E21;
                        else
                                NS_FIFO_Bus <= E22;
                        end if;


                when E21 =>
                        NS_FIFO_Bus <= E9;



                when E22=>                                              -- Send
Least Significant byte of calculated Check Sum (CRC16)
```

```
                STD_FIFO_W_DataIn<= CRC16_Temp_reg_o(7 downto 0);
                STD_FIFO_W_WriteEn <='1';
                XOR_Cnt_rst <= '0';
                NS_FIFO_Bus <= E23;

            when E23=>                                                          -- Send
Most Significant byte of calculated Check Sum (CRC16)
                STD_FIFO_W_DataIn<= CRC16_Temp_reg_o(15 downto 8);
                STD_FIFO_W_WriteEn <='1';
                XOR_Cnt_rst <= '0';
                NS_FIFO_Bus <= S0;
            ---------------------------------------------------------------------------------------------------


            ----------------------------------
            -- Temporary Pass Through - Start
            ----------------------------------

            when S10=>
                RAM_address <= X"01";                                           --
Location of functional code
                NS_FIFO_Bus <= S11;

            when S11=>
                RAM_address <= X"01";                                           --
Location of functional code
                NS_FIFO_Bus <= S16;




            ----------------------------------
            -- Temporary Pass Through - Start
            ----------------------------------




            -- Set Ram_address <= X"01" before this
            when S16=>
                --Statements for Testing
                --STD_FIFO_W_DataIn<= X"DD";
                --STD_FIFO_W_WriteEn <='1';


                --RAM_address <= X"01";
                if(RAM_Data_Out = X"03") then           --Check Cmd (Read) - Original
UART is 0x0F for Read
                    NS_FIFO_Bus <= S17;
                elsif(RAM_Data_Out = X"10") then        --Check Cmd (Write) - Original
UART is 0x0A for Write
                    NS_FIFO_Bus <= S25;
                else                                                            --Check
Cmd (Invalid Data)
                    NS_FIFO_Bus <= S0;
                end if;
```

```
                        RAM_address <= X"00";

                        Rcv_Cnt_rst<='0';

                    -- Start Read Command Sequence


                    when S17=>                                                  --Send
First byte of Packet(Start Deliminator)
                        STD_FIFO_W_DataIn<= RAM_Data_Out;
                        STD_FIFO_W_WriteEn <='1';

                        NS_FIFO_Bus<=S18;


                    when S18=>                                                  --Send
Second byte of Packet (OP_ID)
                        STD_FIFO_W_DataIn <= X"03";             --Send OP_ID (Write)
        Original UART 0X0A, Modbus responds with the same Function Code
                        STD_FIFO_W_WriteEn <='1';

                        RAM_Data_In <= (Reg_Num_L_reg_o + Reg_Num_L_reg_o);
        -- Storing Num More Packets in Local RAM
                        RAM_address <= X"06";       -- Always Local RAM address for Num
More Packets

                        NS_FIFO_Bus<=S19;

                    when S19=>                                                  --Send
Num More Bytes
                        STD_FIFO_W_DataIn <= (Reg_Num_L_reg_o + Reg_Num_L_reg_o);
        --Packet length is the number of data registers

                        RAM_Data_In <= (Reg_Num_L_reg_o + Reg_Num_L_reg_o);
        -- Storing Num More Packets in Local RAM
                        RAM_address <= X"06";       -- Always Local RAM address for Num
More Packets
                        RAM_wea <='1';
                        Ram_Cnt_rst <= '0';


                        STD_FIFO_W_WriteEn <='1';


                        NS_FIFO_Bus<=S20;



                    when S20=>
                        --if(Rcv_Cnt_Out < Reg_Num_L_reg_o + Reg_Num_L_reg_o) then
        --If less than Register Count (Packet Length Register Used Here) -- Bug, changed from <
Num_More_reg_o TO < Reg_Num_L_reg_o + Reg_Num_L_reg_o ON 11/1/2021
                        if(Rcv_Cnt_Out < Reg_Num_L_reg_o) then
                            Bus_Int1_AddrIn <= Base_Addr_reg_o + Rcv_Cnt_Out ; --Send
Address to Bus Interface for Read
```

```
                                Bus_Int1_RE <='1';
--Read Flag to Bus Interface
                                NS_FIFO_Bus<=S21;
                        else
                                Is_Read <= X"01";
-- Sets value to 0, is NOT read
                                LD_Is_Read <= '1';
                                NS_FIFO_Bus<=S100;
                        end if;

                when S21=>
--Wait until data is ready
                        if(Bus_Int1_Busy = '1') then
                                NS_FIFO_Bus<=S21;
                        else
                                NS_FIFO_Bus<=D5;
                        end if;

                when D5=>
                        Temp_Data <= Bus_Int1_DataOut;
                        LD_Temp_Data <= '1';
                        NS_FIFO_Bus<=S22;

                when S22=>
-- Send Data High
                        STD_FIFO_W_DataIn <= Temp_Data_reg_o(15 downto 8);
                        STD_FIFO_W_WriteEn <='1';

                        RAM_Data_In <= Temp_Data_reg_o(15 downto 8);
                        RAM_address <= Rcv_Cnt_Out + Ram_Cnt_Out + 7;     -- +7
                        NS_FIFO_Bus <= D1;

                when D1=>
                        RAM_Data_In <= Temp_Data_reg_o(15 downto 8);
                        RAM_address <= Rcv_Cnt_Out + Ram_Cnt_Out + 7;     -- +7
                        RAM_wea <='1';
                        NS_FIFO_Bus <= D2;

                when D2=>
                        Ram_Cnt_INC <='1';
                        NS_FIFO_Bus <= S23;

                when S23=>
-- Send Data Low
                        STD_FIFO_W_DataIn<= Temp_Data_reg_o(7 downto 0);
                        STD_FIFO_W_WriteEn <='1';

                        RAM_Data_In <= Temp_Data_reg_o(7 downto 0);
                        RAM_address <= Rcv_Cnt_Out + Ram_Cnt_Out + 7;     -- +8


                        NS_FIFO_Bus <= D3;

                when D3=>
                        RAM_Data_In <= Temp_Data_reg_o(7 downto 0);
                        RAM_address <= Rcv_Cnt_Out + Ram_Cnt_Out + 7;     -- +8
```

```
                        RAM_wea <='1';

                        NS_FIFO_Bus <= D4;

                    when D4=>
                        Rcv_Cnt_Inc <='1';

                        NS_FIFO_Bus <= S20;

                    ---------------------------------------------------------------------------------

                    -- Start Write Command Sequence
                    when S25=>
                        -- Using (2 * rcv_cnt_out < Num_More_reg_0) as Num_More is 14 bytes,
but we use 7 registers to hold this: high and low bytes
                        --if(Rcv_Cnt_Out < Num_More_reg_o - 1) then    --If less than Register
Count - CHANGED to Num_More_reg_o from Reg_Cnt_reg_o
                        if(Rcv_Cnt_Out < Reg_Num_L_reg_o) then
                            NS_FIFO_Bus <= S26;
                        else
                            NS_FIFO_Bus <= S402;
        --Done writing, now send Write Response (Confirmation)
                        end if;
                        RAM_address <= Rcv_Cnt_Out + Rcv_Cnt_Out + 7;                -- +1 to
+7

                    when S26=>
                        RAM_address <= Rcv_Cnt_Out + Rcv_Cnt_Out + 7;
                        Temp_Data_High <= RAM_Data_Out;
                        --LD_Temp_Data_High <='1';
                        NS_FIFO_Bus <= S27;

                    when S27=>
                        RAM_address <= Rcv_Cnt_Out + Rcv_Cnt_Out + 7;
                        Temp_Data_High <= RAM_Data_Out;
                        LD_Temp_Data_High <='1';
                        NS_FIFO_Bus <= S28;

                    when S28=>
                        RAM_address <= Rcv_Cnt_Out + Rcv_Cnt_Out + 8;                -- +2 to
+8
                        NS_FIFO_Bus <= S29;

                    when S29=>
                        RAM_address <= Rcv_Cnt_Out + Rcv_Cnt_Out + 8;
                        Temp_Data_Low <= RAM_Data_Out;
                        --LD_Temp_Data_Low <='1';
                        NS_FIFO_Bus <= S30;

                    when S30=>
                        RAM_address <= Rcv_Cnt_Out + Rcv_Cnt_Out + 8;
                        Temp_Data_Low <= RAM_Data_Out;
                        LD_Temp_Data_Low <='1';
                        NS_FIFO_Bus <= S31;

                    when S31=>
```

```
                        NS_FIFO_Bus <= S32;

                when S32=>
                        Bus_Int1_AddrIn <= Base_Addr_reg_o + Rcv_Cnt_Out;   --Send
Address to Bus Interface for Write
                        Bus_Int1_DataIn(15 downto 8)<=Temp_Data_High_reg_o;         --Send
Data to Bus Interface for Write
                        Bus_Int1_DataIn(7 downto 0)<=Temp_Data_Low_reg_o;           --Send
Data to Bus Interface for Write
                        Bus_Int1_WE <='1';
                                --Write Flag to Bus Interface
                        NS_FIFO_Bus <= S33;

                when S33=>
        --Wait until data is ready
                        if(Bus_Int1_Busy = '1') then
                                NS_FIFO_Bus<=S33;
                        else
                                Rcv_Cnt_Inc <='1';
                                NS_FIFO_Bus<=S25;
                        end if;



                ------------------------------------------------------------
                -- Write Response after saving data

                when S402 =>                                                    -- Set Ram
Address to 0X00, Device ID
                        RAM_address <= X"00";
                        NS_Fifo_Bus<=S34;

                -- This needs to be changed to slave ID
                when S34=>                                                      --Send
First byte of Packet(Start Deliminator)
                        STD_FIFO_W_DataIn<= RAM_Data_Out;
                        STD_FIFO_W_WriteEn <='1';
                        NS_FIFO_Bus<=S35;


                when S35=>                                                      --Send
Second byte of Packet (OP_ID)
                        STD_FIFO_W_DataIn <= X"10";                 --Send OP_ID (Write)
        Original UART 0X0A, Modbus responds with the same Function Code
                        STD_FIFO_W_WriteEn <='1';
                        NS_FIFO_Bus<=S36;

                when S36=>                                                      --Send
Fourth byte of Packet (Starting Address High)
                        STD_FIFO_W_DataIn <= Reg_Addr_H_reg_o;  --Send Start Address
High
                        STD_FIFO_W_WriteEn <='1';
                        NS_FIFO_Bus<=S37;

                when S37=>                                                      --Send
Fifth byte of Packet (Starting Address Low)
```

```
                        STD_FIFO_W_DataIn <= Reg_Addr_L_reg_o;   --Send Start Address
Low
                        STD_FIFO_W_WriteEn <='1';
                        NS_FIFO_Bus<=S38;

                    when S38=>                                                  --Send
Sixth byte of Packet (Number of Registers High)
                        STD_FIFO_W_DataIn <= Reg_Num_H_reg_o;   --Send Number of
Registers High
                        STD_FIFO_W_WriteEn <='1';
                        NS_FIFO_Bus<=S39;


                    when S39=>                                                  --Send
Fifth byte of Packet (Number of Registers Low)
                        STD_FIFO_W_DataIn <= Reg_Num_L_reg_o;   --Send Number of
Registers Low
                        STD_FIFO_W_WriteEn <='1';
                        Is_Read <= X"00";                                       -- Sets value to
0, is NOT read
                        LD_Is_Read <= '1';
                        XOR_Cnt_rst <= '0';
                        NS_FIFO_Bus <= S100;


                    ------------------------------------------------------------------------------------


                    -- End Write Command Sequence




                    ------------------------------------------------------------------------
                    -- CRC16 Forming for Responses. Both write and read response will go here
                    ------------------------------------------------------------------------

                    when S100=>                                                 -- Reset
CRC16_Temp_reg_o and Set A_Poly_reg_o
                        CRC16_Temp <= X"FFFF";                                  -- Load
CRC16_Temp_reg_o with all 1's
                        LD_CRC16_Temp <= '1';
                        A_Poly <= X"A001";                                      -- Load A_Poly
                        LD_A_Poly <= '1';
                        CRC_Cnt_rst <= '0';                                     -- Active Low
                        XOR_Cnt_rst <= '0';
                        XOR2_Cnt_rst <= '0';
                        Ram_address <= X"00";                               -- Reset RAM Address
to 0. First data point at address 1, next state increments to 1
                        NS_FIFO_Bus <= S101;

                    when S101=>                                                 -- Check if there
is more data to process
```

```
                        Ram_address <= XOR_Cnt_Out;
                        NS_FIFO_Bus <= S135;

                when S135 =>
                        Ram_address <= XOR_Cnt_Out;
                        NS_FIFO_Bus <= S102;

                when S102=>                                             -- XOR
CRC16_Temp with Data Register (16 bit XOR 8 bit), XOR's one data register with CRC16_Temp
                                CRC16_Temp <=  CRC16_Temp_reg_o XOR (X"00" &
RAM_Data_Out); -- LEFT OF HERE
                                LD_CRC16_Temp <= '1';
                                NS_FIFO_Bus <= S131;

                when S131=>
                        if(CRC16_Temp_reg_o(0) = '1') then
                                Carry_Over <= X"01";
                                LD_Carry_Over <= '1';
                        end if;
                        NS_FIFO_Bus <= S103;


                when S103=>                                             -- Shift
CRC16_Temp_reg_o to the right
                        CRC16_Temp <= '0' & CRC16_Temp_reg_o(15 downto 1);
                        LD_CRC16_Temp <= '1';
                        NS_FIFO_Bus <= S104;

                when S104 =>                                            -- Check for Carry Over
in CRC16_Temp_reg_o
                        if(Carry_Over_reg_o = X"01") then
                                NS_FIFO_Bus <= S105;
                        else
                                NS_FIFO_Bus <= S106;
                        end if;

                when S105 =>                                            -- Carry Over was
positive, CRC16 XOR POLY
                                CRC16_Temp <= CRC16_Temp_reg_o xor A_Poly_reg_o;
                                LD_CRC16_Temp <= '1';

                                Carry_Over <= X"00";
                                LD_Carry_Over <= '1';
                                NS_FIFO_Bus <= S106;

                when S106 =>                                            -- Increment CRC_Cnt
                        CRC_Cnt_INC <= '1';
                        NS_FIFO_Bus <= S120;

                when S120 =>
                        NS_FIFO_Bus <= S107;

                when S107 =>                                            -- Check if CRC_Cnt >
7
                        if(CRC_Cnt_Out > 7) then
                                NS_FIFO_Bus <= S108;
```

```
                else
                        NS_FIFO_Bus <= S131;
                end if;
                Ram_address <= XOR_Cnt_Out;


        when S108 =>
                if(Is_Read_reg_o = X"00") then          -- Write
                        NS_FIFO_Bus <= S109;
                elsif(Is_Read_reg_o = X"01") then
                        NS_FIFO_Bus <= S133;            -- Read
                else
                        NS_FIFO_Bus <= S0;              -- Somethign went wrong
                end if;
                CRC_Cnt_rst <= '0';
                Ram_address <= XOR_Cnt_Out;


        when S109 =>
                if(XOR_Cnt_Out = X"05") then
                        NS_FIFO_Bus <= S112;            -- End of write message
                else
                        XOR_Cnt_INC <= '1';
                        NS_FIFO_Bus <= S110;
                end if;

        when S110 =>
                Ram_Address <= XOR_Cnt_Out;
                NS_FIFO_Bus <= S130;

        when S130 =>
                Ram_Address <= XOR_Cnt_Out;
                NS_FIFO_Bus <= S102;

        when S133 =>
                Ram_Address <= XOR_Cnt_Out;
                NS_FIFO_Bus <= S111;


        when S111 =>
                if(XOR_Cnt_Out = X"01") then
                                NS_FIFO_Bus <= S140;
                elsif(XOR_Cnt_Out < (Reg_Num_L_reg_o + Reg_Num_L_reg_o + 6))
then
                        XOR_Cnt_INC <= '1';
                        NS_FIFO_Bus <= S136;
                else
                        NS_FIFO_Bus <= S112;
                end if;


        when S136 =>
                NS_FIFO_Bus <= S101;

        when S140 =>
                XOR_Cnt_INC <= '1';
```

```
                NS_FIFO_Bus <= S141;

            when S141 =>
                XOR_Cnt_INC <= '1';
                NS_FIFO_Bus <= S142;

            when S142 =>
                XOR_Cnt_INC <= '1';
                NS_FIFO_Bus <= S143;

            when S143 =>
                XOR_Cnt_INC <= '1';
                NS_FIFO_Bus <= S144;

            when S144 =>
                XOR_Cnt_INC <= '1';
                NS_FIFO_Bus <= S145;

            when S145 =>                                        -- Timing
                NS_FIFO_Bus <= S101;



            ------------------------------------------------------------


            ------------------------------------------------------------

            when S112=>                                                 -- Send
Least Significant byte of calculated Check Sum (CRC16)
                STD_FIFO_W_DataIn<= CRC16_Temp_reg_o(7 downto 0);
                STD_FIFO_W_WriteEn <='1';
                XOR_Cnt_rst <= '0';
                NS_FIFO_Bus <= S113;

            when S113=>                                                 -- Send
Most Significant byte of calculated Check Sum (CRC16)
                STD_FIFO_W_DataIn<= CRC16_Temp_reg_o(15 downto 8);
                STD_FIFO_W_WriteEn <='1';
                XOR_Cnt_rst <= '0';
                NS_FIFO_Bus <= D9;



            -- Sequence for 3.5 Char Delay before response message sent
            when D9 =>
                XOR2_Cnt_rst<='0';
                NS_FIFO_Bus <= D10;

            when D10 =>
                if(XOR2_Cnt_Out < 52500) then           -- 52500 = 3.675 Char @ 9600
Baud, minimum of 3.5 Char
                    XOR2_Cnt_INC <= '1';
                    NS_FIFO_Bus<= D11;
                else
                    NS_FIFO_Bus <= S0;
```

```
                                XOR2_Cnt_rst<='0';
                        end if;

                When D11 =>
                        NS_FIFO_Bus <= D10;




                ------------------------------------------------------------------------
                -- End of CRC16 for Responses


                when others =>
                        NS_FIFO_Bus <=S0;
        end case;
end process;
----End Next State Logic for FIFO to Bus
----UART Clock Divider
UART_Clk: process
begin
        wait until clk'event and clk = '1';
        --Synchronize async signal
        rs232_rcv_t<=rs232_rcv;                                 --Synchro1 rs232_rcv
        rs232_rcv_s<=rs232_rcv_t;                       --Synchro2 rs232_rcv

        if(rst = '0' or (busy_reg_o = '0' and busy2_reg_o = '0')) then
                uartclk <='0';
                i <= CONV_STD_LOGIC_VECTOR(CN,16);
        elsif( i = CM ) then
                        uartclk <= '1';
                        i <= X"0000";
        else
                        i <= i+1;
                        uartclk<='0';
                end if;
end process;
---- End UART Clock Divider
----UART_Read
UART_Read: process
begin
        wait until clk'event and clk = '1';
        if rst ='0' or rx_reg_o='0' then
                temp_rcv<= x"00";
                j<=x"0000";
                rx_done<='0';
        elsif rx_reg_o='1' then
                if uartclk='1' then
                        if j<X"09" then
                                temp_rcv(7)<=rs232_rcv_s;
                                temp_rcv(6 downto 0)<=temp_rcv(7 downto 1);
                                j<=j+1;
                                rx_done <='0';
                        else
                                j <= X"0000";
```

```
                                        rx_done <='1';
                                end if;
                        else
                                rx_done<='0';
                        end if;
                end if;
        end process;
        ----End UART_Read
        -----UART_Xmit
        UART_Xmit: process
        begin
                wait until clk'event and clk = '1';
                if (rst = '0' or tx_reg_o='0') then
                        rs232_xmt<='1';
                        tx_done <='0';
                        u<=0;

                        --structure the 10-bit frame to be sent
                        txbuff(9)<='1'; --stopbit 2
                        txbuff(8 downto 1) <= temp2_reg_o;
                        txbuff(0)<='0';    --startbit 2
                else
                        if uartclk = '1' then
                                if(u<10) then
                                        rs232_xmt<= txbuff(0);
                                        txbuff(8 downto 0) <= txbuff(9 downto 1);
                                        tx_done<='0';
                                        u<=u+1;
                                else
                                        u<=0;
                                        tx_done<='1';
                                end if;
                        end if;
                end if;
        end process;
        -----End UART_Xmit


        ----State Sync
        sync_States: process
        begin
                wait until clk'event and clk = '1';
                if rst = '0' then
                        CS_RS232_R <= S0;
                        CS_RS232_W <= S0;
                        CS_FIFO_Bus <= S0;
                else
                        CS_RS232_R <= NS_RS232_R;
                        CS_RS232_W <= NS_RS232_W;
                        CS_FIFO_Bus <= NS_FIFO_Bus;
                end if;
        end process;
        ----End State Sync


end Behavioral;
```

A-2 Digital Twin Top File With Modbus RTU Instantiation: FPGA VHDL Code

```
----------------------------------------------------------------------------------
-- Company:  University of Arkansas (NCREPT)
-- Engineer: Estefano Soria and Paulo Custodio
--
-- Create Date:                26/10/2021
-- Project Name:               Digital_Twin
-- Module Name:                Top
-- Design Name:                Digital_Twin_Top
-- Target Devices:             LCMXO2-7000HC-4FG484C (UCB v1.4a)
-- Tool versions:              Lattice Diamond_x64 Build 3.11
-- Description:
-- This project has the purpose to create a Digital Twin (DT) able to emulate an Active-Neutral Point Clamped (ANPC) inverter using the inactive/standby DSP outputs
-- to check if the new DSP firmware has all the requirements designed with the Design-For-Trust (DFTr) technique.
-- The ANPC inverter has 6 transistors per phase, being two Fast Frequency Transistors(Q2 and Q3) and four Slow Frequency Transistors (Q1,Q4,Q5,Q6).
--
--
-- The strategy used to control the ANPC inverter is considering the PWM 01, which controls the transistor 1 (Q1) the same as Q6, since they must be on and off at the same time.
-- The same concept was applied to transistors Q4 and Q5, because they also have the same behavior.
-- Slow transistors must have a switching frequency equivalent to the fundamental frequency 60Hz, while the Fast transistors (Q2 and Q3) must have a switching frequency of 42kHz.
--
-- PinOut:
-- ------Inputs------
-- ----DSP 1 (DIMM-B)----
-- --Phase A--
-- F20 -> Q1|Q6
-- M16 -> Q2
-- C22 -> Q3
-- K20 -> Q4|Q5
-- --Phase B--
-- G18 -> Q1|Q6
-- M19 -> Q2
-- C21 -> Q3
-- K18 -> Q4|Q5
-- --Phase C--
-- K22 -> Q1|Q6
-- L22 -> Q2
-- B22 -> Q3
-- J17 -> Q4|Q5
-- ---- END DSP 1 (DIMM-B)----
-- ----DSP 2 (DIMM-C)----
-- --Phase A--
-- AB6 -> Q1|Q6
-- Y7  -> Q2
-- T8  -> Q3
-- U10 -> Q4|Q5
-- --Phase B--
-- Y4  -> Q1|Q6
-- V8  -> Q2
-- U8  -> Q3
```

```
-- W11 -> Q4|Q5
-- --Phase C--
-- T10  -> Q1|Q6
-- W9   -> Q2
-- AA8  -> Q3
-- V11  -> Q4|Q5
-- ---- END DSP 2 (DIMM-C)----
-- ----Others----
-- C2  -> Flash SPI Slave Output
-- Y1  -> SCI RX
-- V13 -> SCI RX DSP
-- R3  -> SCI RX WEBSERVER
-- G13 -> Push Button (SW1) -> Erase Flash Memory manually
-- ---- End Others----
-- ------End Inputs------
-- ------Outputs------
-- --Phase A--
-- A21 -> Q1
-- C19 -> Q2
-- A20 -> Q3
-- D18 -> Q4
-- B19 -> Q5
-- C18 -> Q6
-- --Phase B--
-- F17 -> Q1
-- A18 -> Q2
-- D17 -> Q3
-- E17 -> Q4
-- A17 -> Q5
-- C18 -> Q6
-- --Phase C--
-- F16 -> Q1
-- E16 -> Q2
-- D16 -> Q3
-- B15 -> Q4
-- C16 -> Q5
-- E15 -> Q6
-- --LEDs--
-- R17 -> LED A
-- U17 -> LED B
-- T18 -> LED C
-- R16 -> LED D
-- T17 -> LED E
-- Y21 -> LED F
-- Y20 -> LED G
-- U18 -> LED H
-- --Flash Memory--
-- C3  -> Chip-Select (CSSPIN)
-- E4  -> Hold
-- D3  -> SPI Clock (MCLK)
-- F5  -> Slave Input SPI (SISPI)
-- F6  -> Write enable (WPn)
-- --DSPs--
-- G22 -> DIMM_B_GPIO30
-- H16 -> DIMM_B_SCI_RX
-- Y3  -> DIMM_C_GPIO30
```

```
-- AB2 -> DIMM_C_SCI_RX
-- AA22-> IDC_B_GPIO_00 (DSP1 Reset)
-- Y14 -> IDC_C_GPIO_00 (DSP2 Reset)
-- -- Others --
-- AA1 -> SCI_TX
-- V12 -> SCI_TX_DSP
-- R2 -> SCI_TX_Webserver
--
-- Revision:
--          v2.15.22 - Top file without the deadtime component (deadtime should be called by the firmware
validation only)
--          v3.24.22 - Added the emulation control and debug signals. Adapted to ANPC inverter
--          v5.23.22 - Polishment and comments to make it easier to comprehend the code for future work.
-- Additional Comments:
--
--
----------------------------------------------------------------------------------

Library IEEE;
use IEEE.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.numeric_std.all;
library machxo2;
use machxo2.all;
library work;
use work.Digital_Twin_Common.all;
entity Digital_Twin_and_Hot_Patching is
    Port
          (
                    ------------------- Communication pins between CPLD and UI          -------------------
                    SCI_RX : in std_logic;    --UART RX pin for Serial Comm with UI: UI(TX) -> CPLD (RX) ->
Y1
                    SCI_TX : out std_logic;  --UART TX pin for Serial Comm with UI: UI(RX) -> CPLD (TX) ->
AA1
                    SCI_RX_DSP : in std_logic;       --UART RX pin for Serial Comm with UI: UI(TX) -> CPLD
(RX)
                    SCI_TX_DSP : out std_logic;      --UART TX pin for Serial Comm with UI: UI(RX) -> CPLD
(TX)
                    SCI_RX_Webserver : in std_logic;    --UART RX pin for Serial Comm with UI: UI(TX) ->
CPLD (RX) -> R2
       SCI_TX_Webserver : out std_logic;    --UART TX pin for Serial Comm with UI: UI(RX) -> CPLD (TX)
-> R3

                    IDC_B_GPIO_00 : out STD_LOGIC; -- Reset Pin of the DSP DIMM-B -> AA22
                    IDC_C_GPIO_00 : out STD_LOGIC; -- Reset Pin of the DSP DIMM-C -> Y14

                    DIMM_B_SCI_RX : out STD_LOGIC; -- This pin is used to send the firmware to the DSP
and the bootloader. -> H16
                    DIMM_C_SCI_RX : out STD_LOGIC; -- This pin is used to send the firmware to the DSP
and the bootloader. -> AB2
                    DIMM_B_SCI_TX : in STD_LOGIC;  -- This pin is used to send the firmware to the DSP
and the bootloader. -> N/A
                    DIMM_C_SCI_TX : in STD_LOGIC;  -- This pin is used to send the firmware to the DSP
and the bootloader. -> N/A

                    DIMM_B_GPIO30 : out STD_LOGIC; -- G22
```

```
        DIMM_C_GPIO30 : out STD_LOGIC; -- Y3

        Btna : in STD_LOGIC;
        ------------------------------------- INPUTS ---------------------------------------
        -- SW : in STD_LOGIC;
        ------------------- DIMM_B --------------------
        -- Phase A
        DSP1_01_A : in  STD_LOGIC; -- Q1 -> F20 -> DIMM-B_GPIO-12
        DSP1_02_A:  in  STD_LOGIC; -- Q2 -> M16 -> DIMM-B_GPIO-01
        DSP1_03_A : in  STD_LOGIC; -- Q3 -> C22 -> DIMM-B_GPIO-02
        DSP1_04_A : in  STD_LOGIC; -- Q4 -> K20 -> DIMM-B_GPIO-25
        -- Phase B
        DSP1_01_B : IN  STD_LOGIC; -- Q1 -> G18 -> DIMM-B_GPIO-26
        DSP1_02_B : IN  STD_LOGIC; -- Q2 -> M19 -> DIMM-B_GPIO-03
        DSP1_03_B : IN  STD_LOGIC; -- Q3 -> C21 -> DIMM-B_GPIO-04
        DSP1_04_B : IN  STD_LOGIC; -- Q4 -> K18 -> DIMM-B_GPIO-27
        -- Phase C
        DSP1_01_C : IN  STD_LOGIC; -- Q1 -> K22 -> DIMM-B_GPIO-14
        DSP1_02_C : IN  STD_LOGIC; -- Q2 -> L22 -> DIMM-B_GPIO-05
        DSP1_03_C : IN  STD_LOGIC; -- Q3 -> D19 -> DIMM-B_GPIO-06 (DIMM-B_GPIO-00 ->
B22)
        DSP1_04_C : IN  STD_LOGIC; -- Q4 -> J17 -> DIMM-B_GPIO-19

        ------------------- DIMM_C --------------------
        -- Phase A
        DSP2_01_A : in  STD_LOGIC; -- Q1 -> AB6 -> DIMM-C_GPIO-12
        DSP2_02_A:  in  STD_LOGIC; -- Q2 -> Y7 ->  DIMM-C_GPIO-01
        DSP2_03_A : in  STD_LOGIC; -- Q3 -> T8 ->  DIMM-C_GPIO-02
        DSP2_04_A : in  STD_LOGIC; -- Q4 -> U10 -> DIMM-C_GPIO-25
        -- Phase B
        DSP2_01_B : IN  STD_LOGIC; -- Q1 -> Y4 ->  DIMM-C_GPIO-26
        DSP2_02_B : IN  STD_LOGIC; -- Q2 -> V8 ->  DIMM-C_GPIO-03
        DSP2_03_B : IN  STD_LOGIC; -- Q3 -> U8 ->  DIMM-C_GPIO-04
        DSP2_04_B : IN  STD_LOGIC; -- Q4 -> W11 -> DIMM-C_GPIO-27
        -- Phase C
        DSP2_01_C : IN  STD_LOGIC; -- Q1 -> T10 -> DIMM-C_GPIO-14
        DSP2_02_C : IN  STD_LOGIC; -- Q2 -> W9 ->  DIMM-C_GPIO-05
        DSP2_03_C : IN  STD_LOGIC; -- Q3 -> AB7 -> DIMM-C_GPIO-06 (DIMM-C_GPIO-00 ->
AA8)
        DSP2_04_C : IN  STD_LOGIC; -- Q4 -> V11 -> DIMM-C_GPIO-19

        ------------------------------------- OUTPUTS --------------------------------------
        ------------------- Leds --------------------
        LED_A : out STD_LOGIC; -- R17 -> DSP01 is active (D1 from UCB)
        LED_B : out STD_LOGIC; -- U17 -> DSP02 is active (D2 from UCB)
        --LEDs not being used
        LED_C : out STD_LOGIC; -- T18
        LED_D : out STD_LOGIC; -- R16
        LED_E : out STD_LOGIC; -- T17
        LED_F : out STD_LOGIC; -- Y21
        LED_G : out STD_LOGIC; -- Y20
        LED_H : out STD_LOGIC; -- U18

        -- Outputs to control the inverter
        -- Phase A
        SW01_A : out std_logic; -- Q1 -> A21 -> IDC-A_GPIO-00 -> Pin 1
```

```
                SW02_A : out std_logic; -- Q2 -> C19 -> IDC-A_GPIO-01 -> Pin 2
                SW03_A : out std_logic; -- Q3 -> A20 -> IDC-A_GPIO-02 -> Pin 3
                SW04_A : out std_logic; -- Q4 -> D18 -> IDC-A_GPIO-03 -> Pin 4
                SW05_A : out std_logic; -- Q5 -> B19 -> IDC-A_GPIO-04 -> Pin 5
                SW06_A : out std_logic; -- Q6 -> C18 -> IDC-A_GPIO-05 -> Pin 6
                -- Phase B
                SW01_B : out std_logic; -- Q1 -> F17 -> IDC-A_GPIO-06 -> Pin 7
                SW02_B : out std_logic; -- Q2 -> A18 -> IDC-A_GPIO-07 -> Pin 8
                SW03_B : out std_logic; -- Q3 -> D17 -> IDC-A_GPIO-08 -> Pin 9
                SW04_B : out std_logic; -- Q4 -> E17 -> IDC-A_GPIO-09 -> Pin 10
                SW05_B : out std_logic; -- Q5 -> A17 -> IDC-A_GPIO-10 -> Pin 11
                SW06_B : out std_logic; -- Q6 -> C17 -> IDC-A_GPIO-11 -> Pin 12
                -- Phase C
                SW01_C : out std_logic; -- Q1 -> F16 -> IDC-A_GPIO-12 -> Pin 13
                SW02_C : out std_logic; -- Q2 -> E16 -> IDC-A_GPIO-13 -> Pin 14
                SW03_C : out std_logic; -- Q3 -> D16 -> IDC-A_GPIO-14 -> Pin 15
                SW04_C : out std_logic; -- Q4 -> B15 -> IDC-A_GPIO-15 -> Pin 16
                SW05_C : out std_logic; -- Q5 -> C16 -> IDC-A_GPIO-16 -> Pin 17
                SW06_C : out std_logic;  -- Q6 -> E15 -> IDC-A_GPIO-17 -> Pin 18
                -- Debug outputs
                fw_validation_debug_1 : out std_logic; -- Y15  -> IDC-C_GPIO-07 -> Pin 8 (Channel 0 -
Flash_CSSPIN)
                fw_validation_debug_2 : out std_logic; -- AB16 -> IDC-C_GPIO-08 -> Pin 9 (Channel 1 -
Flash_SPISO)
                fw_validation_debug_3 : out std_logic; -- AA16 -> IDC-C_GPIO-09 -> Pin 10 (Channel 2 -
Flash_WPn)
                fw_validation_debug_4 : out std_logic; -- T13  -> IDC-C_GPIO-10 -> Pin 11 (Channel 3 -
Flash_SISPI)
                fw_validation_debug_5 : out std_logic; -- U13  -> IDC-C_GPIO-11 -> Pin 12 (Channel 4 -
Flash_HOLDn)
                fw_validation_debug_6 : out std_logic; -- Y16  -> IDC-C_GPIO-12 -> Pin 13 (Channel 5 -
Flash_MCLK)
                debug_emu_Q1_Q6_A : out std_logic; -- AB15 -> IDC-C_GPIO-01 -> Pin 2
                debug_emu_Q4_Q5_A : out std_logic; -- Y12  -> IDC-C_GPIO-04 -> Pin 5
                debug_emu_Q1_Q6_B : out std_logic; -- W12  -> IDC-C_GPIO-02 -> Pin 3
                debug_emu_Q4_Q5_B : out std_logic; -- V13  -> IDC-C_GPIO-05 -> Pin 6
                debug_emu_Q1_Q6_C : out std_logic; -- V12  -> IDC-C_GPIO-03 -> Pin 4
                debug_emu_Q4_Q5_C : out std_logic; -- AA15 -> IDC-C_GPIO-06 -> Pin 7
                debug_emu : out std_logic;

                Flash_CSSPIN:           inout  std_logic; -- C3 -> CS                -> IDC-D_GPIO-00 ->
Pin 1 (Channel 0)
                Flash_SPISO:   in  std_logic;               -- C2 -> SPO     -> IDC-D_GPIO-01 -> Pin 2
(Channel 1)
                Flash_WPn:              inout  std_logic; -- F6 -> WP                -> IDC-D_GPIO-02 ->
Pin 3 (Channel 2 - Always high)
                Flash_SISPI:    inout  std_logic; -- F5 -> SPI       -> IDC-D_GPIO-03 -> Pin 4 (Channel 3)
                Flash_HOLDn:  inout  std_logic; -- E4 -> Hold      -> IDC-D_GPIO-04 -> Pin 5 (Channel 4
- Always high)
                Flash_MCLK:    inout  std_logic  -- D3 -> clk       -> IDC-D_GPIO-05 -> Pin 6 (Channel 5)

        );
END Digital_Twin_and_Hot_Patching;
ARCHITECTURE Behavioral OF Digital_Twin_and_Hot_Patching is
        -- Oscillator
        SIGNAL  OSC_Stdby : std_logic := '0';
```

```
SIGNAL  OSC_Out : std_logic := '0';
SIGNAL  OSC_SEDSTDBY : std_logic := '0';
-- PLL
--SIGNAL  OSC_Out : std_logic := '0';
SIGNAL  clk : std_logic := '0';
SIGNAL  Pll_Lock : std_logic := '0';
-- Bus Master
SIGNAL Xrqst                    : std_logic := '0';
SIGNAL XDat                             : std_logic := '0';
SIGNAL YDat                             : std_logic := '0';
SIGNAL Data                             : std_logic_vector (15 downto 0) := (others => '0');
SIGNAL Addr                             : std_logic_vector (15 downto 0) := (others => '0');
SIGNAL BusRqst                          : std_logic_vector (9 downto 0) := (others => '0');
SIGNAL BusCtrl                          : std_logic_vector (9 downto 0) := (others => '0');
SIGNAL DSP_RAM_addr                     : std_logic_vector (15 downto 0) := (others => '0');
-- Bootloader
SIGNAL  Bootload_EN : std_logic := '1';
SIGNAL  FW_Type                 : std_logic := '0';
SIGNAL  DSP_rcv         : std_logic := '0';
SIGNAL  DSP_xmt         : std_logic := '0';
SIGNAL  DSP_Rst         : std_logic := '0';

-- Other
SIGNAL  rs232_rcv               : std_logic := '0';
SIGNAL          rs232_xmt               : std_logic := '0';
SIGNAL  Error                   : std_logic := '0';
SIGNAL  Boot_Wrkn               : std_logic := '0';
SIGNAL  Boot_Done               : std_logic := '0';
SIGNAL  HP_EN                           : std_logic := '0';
SIGNAL  HP_Done                 : std_logic := '0';
SIGNAL  Emu_EN                          : std_logic := '0';
SIGNAL  Reset_Cnt_rst           : std_logic := '0';
SIGNAL  Reset_Cnt_INC           : std_logic := '0';
SIGNAL  System_rst              : std_logic := '0';
SIGNAL  Nothing                 : std_logic := '0';
SIGNAL  DSP1_Act                : std_logic := '0';
SIGNAL  DSP1_Act_HP_Out : std_logic := '0';
SIGNAL  DSP_Sync_HP             : std_logic := '0';
SIGNAL  Reset_Cnt_out           : std_logic_vector (7 downto 0) := (others => '0');

SIGNAL          DT_EN : std_logic := '0';
SIGNAL          DT_Rst : std_logic := '0';
SIGNAL          Bad_FW_Bus : std_logic_vector (15 downto 0) := (others => '0');
------------------- DSPs --------------------
--Phase A Inputs
SIGNAL Emu_SW01_A : std_logic := '0';
SIGNAL Emu_SW02_A : std_logic := '0';
SIGNAL Emu_SW03_A : std_logic := '0';
SIGNAL Emu_SW04_A : std_logic := '0';
SIGNAL Emu_SW05_A : std_logic := '0';
SIGNAL Emu_SW06_A : std_logic := '0';
--Phase B Inputs
SIGNAL Emu_SW01_B : std_logic := '0';
SIGNAL Emu_SW02_B : std_logic := '0';
SIGNAL Emu_SW03_B : std_logic := '0';
SIGNAL Emu_SW04_B : std_logic := '0';
```

```
	SIGNAL Emu_SW05_B : std_logic := '0';
	SIGNAL Emu_SW06_B : std_logic := '0';
	--Phase C Inputs
	SIGNAL Emu_SW01_C		: std_logic := '0';
	SIGNAL Emu_SW02_C		: std_logic := '0';
	SIGNAL Emu_SW03_C		: std_logic := '0';
	SIGNAL Emu_SW04_C		: std_logic := '0';
	SIGNAL Emu_SW05_C		: std_logic := '0';
	SIGNAL Emu_SW06_C		: std_logic := '0';
	SIGNAL fw_validation_signal_debug_1 :		std_logic;
	SIGNAL fw_validation_signal_debug_2 :		std_logic;
	SIGNAL fw_validation_signal_debug_3 :		std_logic;
	SIGNAL fw_validation_signal_debug_4 :		std_logic;
	SIGNAL fw_validation_signal_debug_5 :		std_logic;
	SIGNAL fw_validation_signal_debug_6 :		std_logic;


	------------------------------------------ Module Declaration ------------------------------------------
	------------------- Internal Oscillator --------------------
	COMPONENT OSCH
		GENERIC
		(
			NOM_FREQ: string := "8.31"
		);
		PORT
		(
			STDBY :IN std_logic;
			OSC :OUT std_logic;
			SEDSTDBY :OUT std_logic
		);
	END COMPONENT;

	------------------- PLL --------------------
COMPONENT PLL_Clk
		PORT
		(
			ClkI: in  std_logic;
			ClkOP: out std_logic;
			Lock: out std_logic
		);
END COMPONENT;

	------------------- Bus_Master --------------------
COMPONENT Digital_Twin_Bus_Master
	PORT
		(
	clk : IN  std_logic;
	rst : IN  std_logic;
	Data : INOUT  std_logic_vector(15 downto 0);
	Addr : IN  std_logic_vector(15 downto 0);
	Xrqst : IN  std_logic;
	XDat : OUT  std_logic;
	YDat : IN  std_logic;
	BusRqst : IN  std_logic_vector(9 downto 0);
	BusCtrl : OUT  std_logic_vector(9 downto 0);
			Flash_CSSPIN: out  std_logic;
```

```
                        Flash_MCLK: out  std_logic;
                        Flash_SISPI: out  std_logic;
                        Flash_SPISO: in  std_logic;
                        Flash_WPn: out  std_logic;
                        Flash_HOLDn: out  std_logic;
                        Reset_Flash_Button: in std_logic
                );
END COMPONENT;

        ------------------- RS232_Usr_Int -------------------
         COMPONENT RS232_Usr_Int
                Generic
                (
                        Baud            : integer;              -- Baud Rate
                        clk_in          : integer               -- Input Clk
                );

                PORT
                (
        clk : IN  std_logic;
        rst : IN  std_logic;
        rs232_rcv : IN  std_logic;
        rs232_xmt : OUT  std_logic;
        Data : INOUT  std_logic_vector(15 downto 0);
        Addr : OUT  std_logic_vector(15 downto 0);
        Xrqst : OUT  std_logic;
        XDat : IN  std_logic;
        YDat : OUT  std_logic;
        BusRqst : OUT  std_logic;
        BusCtrl : IN  std_logic
   );
END COMPONENT;
        ------------------- Test1_DT_Boot_Ctrl -------------------
        component Digital_Twin_Bootloader_Control
        Port (
                        --IN
                        clk             : in  STD_LOGIC;
                        rst             : in  STD_LOGIC;

                        Data            : INOUT  std_logic_vector(15 downto 0);
                        Addr            : OUT  std_logic_vector(15 downto 0);
                        Xrqst           : OUT  std_logic;
                        XDat            : IN  std_logic;
                        YDat            : OUT  std_logic;
                        BusRqst         : OUT  std_logic;
                        BusCtrl : IN  std_logic;

                        Error           :in STD_LOGIC;
                        Boot_Wrkn       :in STD_LOGIC;
                        Boot_Done       :in STD_LOGIC;
                        HP_EN           :in STD_LOGIC;

                        --OUT
                        Bootload_EN     :out STD_LOGIC;
                        FW_Type                 :out STD_LOGIC;
                        DT_EN           :out STD_LOGIC;
```

```
                        DT_Rst          :out STD_LOGIC


                );
        END component;
        ------------------- DT_Bootloader_Test Component -------------------
        component Digital_Twin_Bootloader
        generic(
                                Baud : integer;                         --9,600 bps
                                clk_in : integer);              --25MHz
Port (
                clk : IN  std_logic;
                rst : IN  std_logic;
                Bootload_EN : IN STD_LOGIC;
                FW_Type         : IN STD_LOGIC;
                Bootload_Wrkn : OUT STD_LOGIC;
                Bootload_Done : OUT STD_LOGIC;

                DSP_rcv : OUT std_logic;
                DSP_xmt : IN  std_logic;
                DSP_Rst : OUT  STD_LOGIC;
                Data : INOUT  std_logic_vector(15 downto 0);
                Addr : OUT  std_logic_vector(15 downto 0);
                Xrqst : OUT  std_logic;
                XDat : IN  std_logic;
                YDat : OUT  std_logic;
                BusRqst : OUT  std_logic;
                BusCtrl : IN  std_logic

        );
        END component;
        ------------------- Std_Counter Component -------------------
        component Std_Counter is
        generic
        (
                Width : integer             -- width of counter
        );
        port(INC,rst,clk: in std_logic;
                 Count: out STD_LOGIC_VECTOR(Width-1 downto 0));
        END component;
        ------------------- DSP_Hot_Patch Component -------------------
        component Digital_Twin_Hot_Patch_Control
Port (
                clk : in std_logic;
                rst : in std_logic;
                EN : in std_logic;
                DSP1_Act_Out : out std_logic;
                DSP_Sync : out std_logic;
                Done : out std_logic
                );
        END component;
        ------------------- Digital_Twin_Emulation_Control -------------------
        COMPONENT Digital_Twin_Emulation_Control
                PORT
                (
                        clk                 : in STD_LOGIC;
```

```
		rst			: in STD_LOGIC;
		Emu_EN			: in std_logic;
		Data			: INOUT  std_logic_vector(15 downto 0);
		Addr			: OUT  std_logic_vector(15 downto 0);
		Xrqst			: OUT  std_logic;
		XDat			: IN  std_logic;
		YDat			: OUT  std_logic;
		BusRqst			: OUT  std_logic;
		BusCtrl : IN  std_logic;
		--Phase A Inputs
		Emu_SW01_A  : in std_logic;
		Emu_SW02_A  : in std_logic;
		Emu_SW03_A  : in std_logic;
		Emu_SW04_A  : in std_logic;
		Emu_SW05_A  : in std_logic;
		Emu_SW06_A  : in std_logic;

		--Phase B Inputs
		Emu_SW01_B  : in std_logic;
		Emu_SW02_B  : in std_logic;
		Emu_SW03_B  : in std_logic;
		Emu_SW04_B  : in std_logic;
		Emu_SW05_B  : in std_logic;
		Emu_SW06_B  : in std_logic;

		--Phase C Inputs
		Emu_SW01_C  : in std_logic;
		Emu_SW02_C  : in std_logic;
		Emu_SW03_C  : in std_logic;
		Emu_SW04_C  : in std_logic;
		Emu_SW05_C  : in std_logic;
		Emu_SW06_C  : in std_logic;
		Error			: in STD_LOGIC;
		HP_EN			: in STD_LOGIC
	);
END component;

	------------------- Test1_DT_Firmware_Validation -------------------
COMPONENT Digital_Twin_Firmware_Validation
	PORT
	(
		clk : in STD_LOGIC;
		rst : in STD_LOGIC;

		Data	: INOUT		std_logic_vector(15 downto 0);
		Addr	: OUT	std_logic_vector(15 downto 0);
		Xrqst	: OUT	std_logic;
		XDat	: IN	std_logic;
		YDat	: OUT	std_logic;
		BusRqst : OUT	std_logic;
		BusCtrl : IN	std_logic;

		--Phase A Inputs
		Emu_SW01_A  : in std_logic;
		Emu_SW02_A  : in std_logic;
		Emu_SW03_A  : in std_logic;
```

```
                        Emu_SW04_A  : in std_logic;
                        Emu_SW05_A  : in std_logic;
                        Emu_SW06_A  : in std_logic;

                        --Phase B Inputs
                        Emu_SW01_B  : in std_logic;
                        Emu_SW02_B  : in std_logic;
                        Emu_SW03_B  : in std_logic;
                        Emu_SW04_B  : in std_logic;
                        Emu_SW05_B  : in std_logic;
                        Emu_SW06_B  : in std_logic;

                        --Phase C Inputs
                        Emu_SW01_C  : in std_logic;
                        Emu_SW02_C  : in std_logic;
                        Emu_SW03_C  : in std_logic;
                        Emu_SW04_C  : in std_logic;
                        Emu_SW05_C  : in std_logic;
                        Emu_SW06_C  : in std_logic;

                        HP_Done             : in std_logic;     -- Signal coming from DSP_Hot-Patch
module saying that hot-patch is completed
                        Boot_Done       : in std_logic; -- Signal to inform that the boot loading is done
                        Boot_Wrkn       : in std_logic; -- Signal to inform that the boot loading is working
                        Emu_EN              : out std_logic; -- Start the emulation
                        HP_EN               : out std_logic; -- Signal sent to DSP_Hot-Patch module
to enable HP PROCESS
                        Bad_FW_Bus      : in std_logic_vector (15 downto 0);  -- Not being used
                        Error               : out std_logic; -- Signal error to stop all other PROCESSes
                        DSP1_Act        : in std_logic
                );
        END COMPONENT;
BEGIN ------------------------------------------------------------ BEGIN ------------------------------------------------------------
        ------------------- Instantiate Internal Oscillator --------------------
        Int_OSC: OSCH PORT MAP (
                STDBY => OSC_Stdby,
                OSC => OSC_Out,
                SEDSTDBY => OSC_SEDSTDBY
        );


        ------------------- Instantiate PLL --------------------
        PLL_1: PLL_Clk PORT MAP (
                ClkI => OSC_Out,
                ClkOP => clk,
                Lock =>Pll_Lock
        );
        ------------------- Instantiate Bus_Master --------------------
        BM: Digital_Twin_Bus_Master PORT MAP (
                clk                     => clk,
                rst                     => System_rst,
                Data                    => Data,
                Addr                    => Addr,
                Xrqst                   => Xrqst,
                XDat                    => XDat,
                YDat                    => YDat,
```

```
                BusRqst                 => BusRqst,
                BusCtrl         => BusCtrl,
                Flash_CSSPIN  => Flash_CSSPIN,
                Flash_MCLK              => Flash_MCLK,
                Flash_SISPI             => Flash_SISPI,
                Flash_SPISO             => Flash_SPISO,
                Flash_WPn               => Flash_WPn,
                Flash_HOLDn             => Flash_HOLDn,
                Reset_Flash_Button => Btna
);

        ------------------- Instantiate RS232_Usr_Int -------------------
        RS232_Usr: RS232_Usr_Int
        Generic Map
         (
                Baud    => 9600,        -- Baud Rate
                Clk_In => Clk_Freq      --      Input Clk
         )
        PORT MAP (
    clk => clk,
    rst => System_rst,
    rs232_rcv => SCI_RX,
    rs232_xmt => SCI_TX,
                --rs232_xmt => Temp_Debug,
    Data => Data,
    Addr => Addr,
    Xrqst => Xrqst,
    XDat => XDat,
    YDat => YDat,
    BusRqst => BusRqst(1), -- Was 3
    BusCtrl => BusCtrl(1) -- Was 3
);

         --DSP
        RS232_Usr_DSP: RS232_Usr_Int
        Generic Map
         (
                Baud    => 9600,        -- Baud Rate
                Clk_In => Clk_Freq      --      Input Clk
         )
        PORT MAP (
    clk => clk,
    rst => System_rst,
    rs232_rcv => SCI_RX_DSP,
    rs232_xmt => SCI_TX_DSP,
                --rs232_xmt => Temp_Debug,
    Data => Data,
    Addr => Addr,
    Xrqst => Xrqst,
    XDat => XDat,
    YDat => YDat,
    BusRqst => BusRqst(2), -- Was 3
    BusCtrl => BusCtrl(2) -- Was 3
);
         --Webserver
        RS232_Usr_Webserver: RS232_Usr_Int
```

```
Generic Map
 (
 Baud     => 9600,    -- Baud Rate
 Clk_In => Clk_Freq    --    Input Clk
 )
PORT MAP (
  clk => clk,
  rst => System_rst,
  rs232_rcv => SCI_RX_Webserver,
  rs232_xmt => SCI_TX_Webserver,
  --rs232_xmt => Temp_Debug,
  Data => Data,
  Addr => Addr,
  Xrqst => Xrqst,
  XDat => XDat,
  YDat => YDat,
  BusRqst => BusRqst(5), -- Was 3
  BusCtrl => BusCtrl(5) -- Was 3
);
------------------- Instantiate Boot_Ctrl --------------------
Boot_Ctrl: Digital_Twin_Bootloader_Control
PORT MAP (
          clk                => clk,
          rst                => System_rst,
          Data               => Data,
          Addr               => Addr,
          Xrqst              => Xrqst,
          XDat               => XDat,
          YDat               => YDat,
          BusRqst            => BusRqst(0),
          BusCtrl => BusCtrl(0),
          Error              => Error,
          Boot_Wrkn          => Boot_Wrkn,
          Boot_Done          => Boot_Done,
          HP_EN              => HP_EN,
          Bootload_EN        => Bootload_EN,
          FW_Type                    => FW_Type,
          DT_EN              => DT_EN,
          DT_Rst             => DT_Rst
);
------------------- Instantiate Bootloader --------------------
Bootload: Digital_Twin_Bootloader
generic map
(
          Baud                       => 9600,                  --9,600 bps
          clk_in                     => Clk_Freq    --25MHz
)
port map (
          clk                                => clk,
          rst                                => System_rst,
          Bootload_EN                => Bootload_EN,
          FW_Type                                    => FW_Type,
          Bootload_Wrkn              => Boot_Wrkn,
          Bootload_Done              => Boot_Done,
          DSP_rcv                            =>        DSP_rcv,
--FW_BIT_OUT,          ---- THIS FW_BIT_OUT SIGNAL IS ONLY USED FOR THIS TEST,
```

```
USUALLY THIS CONNECTS TO THE SERIAL PORT OF THE DSP THROUGH DIMM B OR DIMM C
DEPENDING ON THE DSP, AND IT IS NOW CONNECTED THROUGH THE HP PROCESS BELOW ----
                DSP_xmt                         =>      DSP_xmt,
        --xmt,          ---- THIS xmt SIGNAL IS ONLY FOR THIS TEST, AND NEEDS TO BE
INITIALIZED TO 1 ----
                DSP_Rst                         =>      DSP_Rst,
        --DSP_Rst,          ---- ONLY FOR THIS TEST, USUALLY CONNECTS TO THE
EXTERNAL GPIO PIN THAT IS SOLDERED TO THE DSP TO BE ABLE TO RESET IT, AND IT IS NOW
CONNECTED THROUGH THE HP PROCESS BELOW ----
                Data                            => Data,
                Addr                            => Addr,
                Xrqst                           => Xrqst,
                XDat                            => XDat,
                YDat                            => YDat,
                BusRqst                         => BusRqst(4),
                BusCtrl                 => BusCtrl(4)
        );
        ------------------- Instantiate Reset_Cnt_8 --------------------
        Reset_Cnt: Std_Counter
        generic map
        (
                Width => 8
        )
        port map (
                clk => OSC_Out,
                rst=> Reset_Cnt_rst,
                INC=> Reset_Cnt_INC,
                Count=> Reset_Cnt_out
        );
        ------------------- Instantiate HP -------------------
        HP_Set: Digital_Twin_Hot_Patch_Control
        PORT MAP (
                clk => clk,
                rst => System_rst,
                EN => HP_EN,
                DSP1_Act_Out => DSP1_Act_HP_Out,
                DSP_Sync => DSP_Sync_HP,
                Done => HP_Done
        );

        ------------------- Instantiate Emu_Ctrl -------------------
        Emu_Ctrl: Digital_Twin_Emulation_Control
        PORT MAP (
                clk     => clk,
                rst     => System_rst,

                Emu_EN          => Emu_EN,

                Data    => Data,
                Addr    => Addr,
                Xrqst   => Xrqst,
                XDat    => XDat,
                YDat    => YDat,
                BusRqst  => BusRqst(3),
                BusCtrl  => BusCtrl(3),
                -- Phase A
```

```
        Emu_SW01_A => Emu_SW01_A,
        Emu_SW02_A => Emu_SW02_A,
        Emu_SW03_A => Emu_SW03_A,
        Emu_SW04_A => Emu_SW04_A,
        Emu_SW05_A => Emu_SW05_A,
        Emu_SW06_A => Emu_SW06_A,
        -- Phase B
        Emu_SW01_B => Emu_SW01_B,
        Emu_SW02_B => Emu_SW02_B,
        Emu_SW03_B => Emu_SW03_B,
        Emu_SW04_B => Emu_SW04_B,
        Emu_SW05_B => Emu_SW05_B,
        Emu_SW06_B => Emu_SW06_B,
        -- Phase C
        Emu_SW01_C => Emu_SW01_C,
        Emu_SW02_C => Emu_SW02_C,
        Emu_SW03_C => Emu_SW03_C,
        Emu_SW04_C => Emu_SW04_C,
        Emu_SW05_C => Emu_SW05_C,
        Emu_SW06_C => Emu_SW06_C,
        Error      => Error,
        HP_EN              => HP_EN
);
------------------- Instantiate Firmware Validation_EN --------------------
FW_Valid: Digital_Twin_Firmware_Validation
PORT MAP
(
        clk                => clk,
        rst                => System_rst,
        Data               => Data,
        Addr               => Addr,
        Xrqst              => Xrqst,
        XDat               => XDat,
        YDat               => YDat,
        BusRqst            => BusRqst(6),
        BusCtrl => BusCtrl(6),
        -- Phase A
        Emu_SW01_A => Emu_SW01_A,
        Emu_SW02_A => Emu_SW02_A,
        Emu_SW03_A => Emu_SW03_A,
        Emu_SW04_A => Emu_SW04_A,
        Emu_SW05_A => Emu_SW05_A,
        Emu_SW06_A => Emu_SW06_A,
        -- Phase B
        Emu_SW01_B => Emu_SW01_B,
        Emu_SW02_B => Emu_SW02_B,
        Emu_SW03_B => Emu_SW03_B,
        Emu_SW04_B => Emu_SW04_B,
        Emu_SW05_B => Emu_SW05_B,
        Emu_SW06_B => Emu_SW06_B,
        -- Phase C
        Emu_SW01_C => Emu_SW01_C,
        Emu_SW02_C => Emu_SW02_C,
        Emu_SW03_C => Emu_SW03_C,
        Emu_SW04_C => Emu_SW04_C,
        Emu_SW05_C => Emu_SW05_C,
```

```
            Emu_SW06_C => Emu_SW06_C,
            HP_Done                 => HP_Done,
            Boot_Done       => Boot_Done,
Boot_Wrkn=> Boot_Wrkn,
Emu_EN              => Emu_EN,
HP_EN               => HP_EN,
            Bad_FW_Bus   => Bad_FW_Bus,
Error               => Error,
            DSP1_Act        => DSP1_Act
    );

    ------------------  Oscillator ------------------
    OSC_Stdby <= '0';
    ------------------ Tie unused ports to '0'------------------
    BusRqst(9 downto 7) <= (others => '0');
    Bad_FW_Bus(15 downto 2) <= (others => '0'); -- Obsolete
    ------------------ Reset Block1 ------------------
    Reset_Blk1: PROCESS
    BEGIN
                    wait until OSC_Out'event and OSC_Out = '1';
                            IF (PLL_Lock ='0') THEN
                                    Reset_Cnt_rst <= '0';
                            else
                                    Reset_Cnt_rst <= '1';
                            END IF;
    END PROCESS;
    ------------------ Reset Block ------------------
    Reset_Blk: PROCESS
    BEGIN
                    wait until OSC_Out'event and OSC_Out = '1';
                            IF (Reset_Cnt_out < X"7F") THEN --7F = 127
                                    System_rst <= '0';
                                    Reset_Cnt_INC <='1';
                            else
                                    System_rst <= '1';
                                    Reset_Cnt_INC <='0';
                            END IF;
    END PROCESS;

    ------------------ Setting DSP1 assignment and debug signals ------------------
    DSP1_Act_Set: PROCESS
    BEGIN
            wait until clk'event and clk = '1';
                    IF (System_rst = '0') THEN
                            DSP1_Act <= '1';
                    else
                            DSP1_Act <= DSP1_Act_HP_Out;
                    END IF;
                    fw_validation_debug_1 <= Flash_CSSPIN;
                    fw_validation_debug_2 <= Flash_SPISO;
                    fw_validation_debug_3 <= DSP_rcv;
                    fw_validation_debug_4 <= Flash_SISPI;
                    fw_validation_debug_5 <= Flash_HOLDn;
                    fw_validation_debug_6 <= Flash_MCLK;
                    debug_emu_Q1_Q6_A <= Emu_SW01_A;
                    debug_emu_Q4_Q5_A <= Emu_SW04_A;
```

```
                debug_emu_Q1_Q6_B <= Emu_SW01_B;
                debug_emu_Q4_Q5_B <= Emu_SW04_B;
                debug_emu_Q1_Q6_C <= Emu_SW01_C;
                debug_emu_Q4_Q5_C <= Emu_SW04_C;
END PROCESS;

------------------- Main Routing PROCESS (Combinatorial) -------------------

PROCESS (SCI_RX, DSP_Rst, DSP_rcv, DSP1_Act)
        BEGIN
                IF (DSP1_Act = '1') THEN
                        ------------------- DSP 1 Active -------------------
                        -- Phase A
                        SW01_A <= DSP1_01_A;
                        SW02_A <= DSP1_02_A;
                        SW03_A <= DSP1_03_A;
                        SW04_A <= DSP1_04_A;
                        SW05_A <= DSP1_04_A; -- Same as PWM 04
                        SW06_A <= DSP1_01_A; -- Same as PWM 01
                        -- Phase B
                        SW01_B <= DSP1_01_B;
                        SW02_B <= DSP1_02_B;
                        SW03_B <= DSP1_03_B;
                        SW04_B <= DSP1_04_B;
                        SW05_B <= DSP1_04_B; -- Same as PWM 04
                        SW06_B <= DSP1_01_B; -- Same as PWM 01
                        -- Phase C
                        SW01_C <= DSP1_01_C;
                        SW02_C <= DSP1_02_C;
                        SW03_C <= DSP1_03_C;
                        SW04_C <= DSP1_04_C;
                        SW05_C <= DSP1_04_C; -- Same as PWM 04
                        SW06_C <= DSP1_01_C; -- Same as PWM 01

                        ------------------- DSP 2 Emulation -------------------
                        -- Phase A
                        Emu_SW01_A <= DSP2_01_A;
                        Emu_SW02_A <= DSP2_02_A;
                        Emu_SW03_A <= DSP2_03_A;
                        Emu_SW04_A <= DSP2_04_A;
                        Emu_SW05_A <= DSP2_04_A; -- Same as PWM 04
                        Emu_SW06_A <= DSP2_01_A; -- Same as PWM 01
                        -- Phase B
                        Emu_SW01_B <= DSP2_01_B;
                        Emu_SW02_B <= DSP2_02_B;
                        Emu_SW03_B <= DSP2_03_B;
                        Emu_SW04_B <= DSP2_04_B;
                        Emu_SW05_B <= DSP2_04_B; -- Same as PWM 04
                        Emu_SW06_B <= DSP2_01_B; -- Same as PWM 01
                        -- Phase C
                        Emu_SW01_C <= DSP2_01_C;
                        Emu_SW02_C <= DSP2_02_C;
                        Emu_SW03_C <= DSP2_03_C;
                        Emu_SW04_C <= DSP2_04_C;
                        Emu_SW05_C <= DSP2_04_C; -- Same as PWM 04
                        Emu_SW06_C <= DSP2_01_C; -- Same as PWM 01
```

```
                    ------------------------------------------------------------
                    DIMM_C_GPIO30 <= DSP_Sync_HP;
                    DIMM_B_GPIO30 <= DSP_Sync_HP;
                    LED_A <= '0';
                    LED_B <= '1';

                    LED_C <= '1';
                    LED_D <= '1';
                    LED_E <= '1';
                    LED_F <= '1';
                    LED_G <= '1';
                    LED_H <= '1';

                    IDC_B_GPIO_00 <= '1';                   ---- Reset is active low, and
1(NO Reset) is routed to pin 00 of IDC B (DSP1 is Active)
                    IDC_C_GPIO_00 <= DSP_Rst;           ---- DSP_Rst signal(Bootloader)
routed to pin 00 of IDC C (DSP2 is Stand-By)

                    DIMM_B_SCI_RX <= '1';                       ---- Stop bit is high, and
is sent to the serial receiver of DIMM B (DSP1 is Active)
                    DIMM_C_SCI_RX <= DSP_rcv;       ---- DSP_rsv signal(Bootloader) is
routed to the serial receiver of DIMM C (DSP2 is Stand-By)

                    Nothing <= DIMM_B_SCI_TX;
                    DSP_xmt <= DIMM_C_SCI_TX;
                ELSE
                    ------------------- DSP 2 Active --------------------
                    -- Phase A
                    SW01_A <= DSP2_01_A;
                    SW02_A <= DSP2_02_A;
                    SW03_A <= DSP2_03_A;
                    SW04_A <= DSP2_04_A;
                    SW05_A <= DSP2_04_A;
                    SW06_A <= DSP2_01_A;
                    -- Phase B
                    SW01_B <= DSP2_01_B;
                    SW02_B <= DSP2_02_B;
                    SW03_B <= DSP2_03_B;
                    SW04_B <= DSP2_04_B;
                    SW05_B <= DSP2_04_B;
                    SW06_B <= DSP2_01_B;
                    -- Phase C
                    SW01_C <= DSP2_01_C;
                    SW02_C <= DSP2_02_C;
                    SW03_C <= DSP2_03_C;
                    SW04_C <= DSP2_04_C;
                    SW05_C <= DSP2_04_C;
                    SW06_C <= DSP2_01_C;
                    ------------------- DSP 1 Emulation --------------------
                    -- Phase A
                    Emu_SW01_A <= DSP1_01_A;
                    Emu_SW02_A <= DSP1_02_A;
                    Emu_SW03_A <= DSP1_03_A;
                    Emu_SW04_A <= DSP1_04_A;
                    Emu_SW05_A <= DSP1_04_A;
```

```
                                    Emu_SW06_A <= DSP1_01_A;
                                    -- Phase B
                                    Emu_SW01_B <= DSP1_01_B;
                                    Emu_SW02_B <= DSP1_02_B;
                                    Emu_SW03_B <= DSP1_03_B;
                                    Emu_SW04_B <= DSP1_04_B;
                                    Emu_SW05_B <= DSP1_04_B;
                                    Emu_SW06_B <= DSP1_01_B;
                                    -- Phase C
                                    Emu_SW01_C <= DSP1_01_C;
                                    Emu_SW02_C <= DSP1_02_C;
                                    Emu_SW03_C <= DSP1_03_C;
                                    Emu_SW04_C <= DSP1_04_C;
                                    Emu_SW05_C <= DSP1_04_C;
                                    Emu_SW06_C <= DSP1_01_C;

                                    ------------------------------------------------------
                                    DIMM_C_GPIO30 <= DSP_Sync_HP;
                                    DIMM_B_GPIO30 <= DSP_Sync_HP;

                                    LED_A <= '1';
                                    LED_B <= '0';
                                    LED_C <= '1';
                                    LED_D <= '1';
                                    LED_E <= '1';
                                    LED_F <= '1';
                                    LED_G <= '1';
                                    LED_H <= '1';

                                    IDC_B_GPIO_00 <= DSP_Rst;           ---- DSP_Rst signal(Bootloader)
routed to pin 00 of IDC B (DSP1 is Stand-By)
                                    IDC_C_GPIO_00 <= '1';           ---- Reset is active low, and 1(NO
Reset) is routed to pin 00 of IDC C (DSP2 is Active)

                                    DIMM_B_SCI_RX <= DSP_rcv;           ---- DSP_rsv signal(Bootloader)
is routed to the serial receiver of DIMM B (DSP1 is Stand-By)
                                    DIMM_C_SCI_RX <= '1';           ---- Stop bit is high, and is sent to the
serial receiver of DIMM C (DSP2 is Active)

                                    DSP_xmt <= DIMM_B_SCI_TX;
                                    Nothing <= DIMM_C_SCI_TX;
                                END IF;
        END PROCESS;
END Behavioral;
```

A-3 Python Resiliency Testing Code

```
# Company:  University of Arkansas
# Engineer: Brady McBride
# Project:  SETO embedded Modbus RTU testing
# Version:  2.0
#
#
# 26Feb2022
# Description:  This program is used to test the minimum timing in between Modbus RTU calls within the
#               Modbus RTU VHDL implementation along with the accuracy of dynamically changing data
through
#               read and write commands


# imports
import minimalmodbus
import serial
import time

# instr creation - This is COM10, or /dev/ttyS9
# https://stackoverflow.com/questions/67303888/minimalmodbus-i-can-read-data-but-i-cannot-write-it-rtu
# https://stackoverflow.com/questions/38444497/trouble-locating-my-serial-ports-using-bash-on-windows-
10
instr = minimalmodbus.Instrument('COM6', 1)  # port name, slave address (in decimal)

# instr config
instr.serial.baudrate = 9600
instr.serial.timeout = 4  # seconds
instr.mode = minimalmodbus.MODE_RTU
instr.serial.parity = serial.PARITY_NONE  # not using
instr.serial.bytesize = 8
instr.serial.stopbits = 1
# instr.clear_buffers_before_each_transaction = True
#
instr.byteorder = minimalmodbus.BYTEORDER_LITTLE
#
instr.debug = True

RegList = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28,
29,
        30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55,
56,
        57, 58, 59, 60, 61, 62, 63, 64]

Reglist2 = []

count1 = 0

readLap = 0
writeLap = 0
gapLap = 0
gapLap2 = 0
gapTime2 = 0
gapTime = 0
errorTime = 0
```

```
errorCount = 0
passCount = 0
laps = 10000

start = time.time_ns()
while (count1 < laps):

    # Writing Test
    try:
        print("WRITE")
        gapLap = gapLap + time.time_ns() - gapTime
        writeTime = time.time_ns()
        instr.write_registers(0X100, RegList)
        writeLap = writeLap + time.time_ns() - writeTime
        errorTime = time.time_ns() - writeTime
        gapTime2 = time.time_ns()
        if errorTime > 3000000000:  # 3 seconds
            errorCount = errorCount + 1
            errorTime = 0
    except Exception as e:
        print("Error writing Coil: ", e)


    # Reading Test
    try:
        if gapTime2 > 0:
            gapLap2 = gapLap2 + time.time_ns() - gapTime2
        print("READ:")
        readTime = time.time_ns()

        RegList2 = instr.read_registers(0X100, 64)
        print("RegList2 = ", RegList2)
        if RegList2 != RegList:
            errorCount = errorCount + 1
        else:
            passCount = passCount + 1


        readLap = readLap + time.time_ns() - readTime
        errorTime = time.time_ns() - readTime
        gapTime = time.time_ns()
        print("Error Time = ", errorTime)
        if errorTime > 3000000000:  # 3 seconds
            errorCount = errorCount + 1
            errorTime = 0
    except Exception as e:
        print("Error reading register: ", e)

    # Changing register values
    RegList = [x + 1 for x in RegList]  # Increments every value in the list by 1

    count1 = count1 + 1
    print("Count1 = ", count1)
    print("Error Count = ", errorCount)
```

```
end = time.time_ns()
total = end - start
avg = total / count1
print("Average Time Between Send to Send is ", avg, "ns")
print("    OR    ")
print(avg / 2000, "us   OR   ", avg / 2000000, "ms")

print("\n\nResults from ", laps, " read and ", laps, " write commands:")
print("-------------------------------------------------\n")
print("Total Errors Received:  ", errorCount)
print("Total Passes Received:   ", passCount, "\n")

print("Average Lap Time = ", avg / 2000000, "ms")
print("Average Read Lap Time = ", readLap / (count1 * 1000000))
print("Average Write Lap Time = ", writeLap / (count1 * 1000000))
# print("Gap Time 1 = ", gapLap/(count1*1000000))
# print("Gap Time 2 = ", gapLap2/(count1*1000000))

# time.sleep(1)
```

A-4 CRC Verification with C Code

```
#Code to verify CRC calculation algorithm and output in a non-VHDL environment
#include <iostream>

using namespace std;

int main()
{
   int ram_Size = 6;
   int n = 0;
   int k = 0;
   int CRC16 = 0xFFFF;
   int ram[ram_Size] = {0x01, 0x03, 0x01, 0x00, 0x00, 0x04};
   int poly = 0xa001;
   bool carry = false;


   while(k < ram_Size){
      cout << "k = " << k << endl;
      cout << "ram[" << k << "] = ";
      cout << hex << ram[k] << endl;

      CRC16 = CRC16 ^ ram[k];
      cout << hex << CRC16 << endl;

      n = 0;

      while(n < 8){
         cout << "n = " << n << endl;
         if(CRC16 % 2 == 1)
            carry = true;
         (CRC16 = CRC16 >> 1);
         cout << hex << CRC16 << endl;
         if(carry){
            CRC16 = CRC16 ^ poly;
            cout << "POLY" << endl;
            cout << hex << CRC16 << endl;
         }
         carry = false;
         n++;
      }
      k++;

      if(CRC16 == 0x0FDA){
         cout << "CRC Is Correct" << endl;
   }
   }

   cout << hex << CRC16 << endl;
   //0x4763



   return 0;
}
```