UNIBUS: A UNIVERSAL HARDWARE ARCHITECTURE FOR SERIAL BUS INTERFACES WITH REAL-TIME SUPPORT

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES
OF
MIDDLE EAST TECHNICAL UNIVERSITY

BY

MEHDI DUMAN

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF MASTER OF SCIENCE
IN
ELECTRICAL AND ELECTRONICS ENGINEERING

JANUARY 2015

Approval of the thesis:

UNIBUS: A UNIVERSAL HARDWARE ARCHITECTURE FOR SERIAL BUS INTERFACES WITH REAL-TIME SUPPORT

submitted by **MEHDI DUMAN** in partial fulfillment of the requirements for the degree of **Master of Science in Electrical and Electronics Engineering Department, Middle East Technical University** by,

Prof. Dr. Gülbin Dural Ünver
Dean, Graduate School of **Natural and Applied Sciences** _____

Prof. Dr. Gönül Turhan Sayan
Head of Department, **Electrical and Electronics Engineering** _____

Assoc. Prof. Dr. Şenan Ece Güran Schmidt
Supervisor, **Electrical and Electronics Eng. Dept., METU** _____

Examining Committee Members:

Prof. Dr. Gözde Bozdağı Akar
Electrical and Electronics Engineering Dept., METU _____

Assoc. Prof. Dr. Şenan Ece Güran Schmidt
Electrical and Electronics Engineering Dept., METU _____

Assoc. Prof. Dr. Cüneyt F. Bazlamaçcı
Electrical and Electronics Engineering Dept., METU _____

Dr. Nizam Ayyıldız
ASELSAN,REHİS _____

Dr. Salih Zengin
TÜBİTAK SAGE _____

Date: _____

I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.

Name, Last Name: MEHDI DUMAN

Signature :

ABSTRACT

UNIBUS: A UNIVERSAL HARDWARE ARCHITECTURE FOR SERIAL BUS INTERFACES WITH REAL-TIME SUPPORT

Duman, Mehdi

M.S., Department of Electrical and Electronics Engineering

Supervisor : Assoc. Prof. Dr. Şenan Ece Güran Schmidt

January 2015, 61 pages

Serial bus communication is widely used in different application areas such as Ethernet in computer networking, CAN bus in in-vehicle communications, MIL-STD 1553B in military avionics and UART for peripheral device communication. This thesis work presents UNIBUS (Universal Bus); an abstract, generic block level hardware architecture for implementing serial bus interfaces. UNIBUS realizes the physical and data link layer functions supporting the strict timing requirements for bit operations and synchronization.

The hardware blocks and signal interfaces among these blocks are designed to separate the protocol specific and protocol independent components to increase reusability. A specific serial bus protocol can be implemented using UNIBUS by defining the protocol specific operations and interfaces.

The versatility of UNIBUS is demonstrated by realizing CAN, UART, ARINC-708, ARINC-717 and MIL-STD-1553B on this architecture. These serial bus interfaces are purposely selected to be from different application areas and levels of complexity. All these interfaces are implemented using MODELSIM simulation tool and tested by realizing a sender and receiver that exchange messages as specified. Furthermore MIL-STD- 1553B is fully implemented on FPGA and its correctness is verified by communication to a commercial chip. The analysis of the resource and power con-

sumption of the realizations shows that the generality of the architecture does not decrease the efficiency of the implementations.

UNIBUS decreases the hardware development time for existing and possibly new serial bus protocols by providing the readiliy designed blocks and signal interfaces. Furthermore UNIBUS increases the reliability of the design as the reused protocol independent components that are common among different protocols need to be verified only once and the blocks together with their interfaces are clearly defined. UNIBUS can be both used for the development of full scale serial bus interface components to be used in real systems as well as developing test benches for existing products. In such deployment, a given bus interface's desired functions can be implemented on UNIBUS to achieve a communicating counterpart for the tested component.

Keywords: Real time, serial, embedded bus protocol, generic architecture, encoder, decoder, field programmable gate array (FPGA), hardware, universal bus, serial bus

ÖZ

UNIBUS: SERİ VERİYOLU ARAYÜZLERİ İÇİN GERÇEK ZAMAN DESTEKLİ GENEL BİR DONANIM MİMARİSİ

Duman, Mehdi
Yüksek Lisans, Elektrik ve Elektronik Mühendisliği Bölümü
Tez Yöneticisi : Doç. Dr. Şenan Ece Güran Schmidt

Ocak 2015 , 61 sayfa

Seri veriyolu haberleşmesi genel olarak, bilgisayar ağlarındaki Ethernet veriyolu, araç içi haberleşmesindeki CAN veriyolu, havacılıkdaki MIL-STD-1553B veriyolu, çevre birimi cihaz haberleşmesindeki UART veriyolu gibi uygulamalarda kullanılmaktadır. Bu tez çalışması, seri veriyolu arayüzleri uygulamaları için UNIBUS (genel veriyolu); blok seviyesinde öz ve genel bir donanım mimarisi sunmaktadır. UNIBUS, bit operasyonları ve bit senkronizasyonu için gerekli olan katı zamanlama ihtiyaçlarını destekleyerek veri bağı ve fiziksel katman fonksiyonlarını gerçeklemektedir.

Donanım blokları ve bu bloklar arasındaki sinyal arayüzleri, yeniden kullanılabilirliği artırmak için protokole özel ve protokolden bağımsız bileşenleri ayıracak şekilde tasarlanmaktadır. Belirli bir seri veriyolu protokolü, protokole özgü operasyonlar ve arayüzler tanımlanarak UNIBUS ile gerçeklenebilir.

UNIBUS mimarisinin çok yönlülüğü, CAN, UART, ARINC-708, ARINC-717 and MIL-STD-1553B protokolleri gerçeklenerek gösterilmektedir. Bu protokoller, farklı uygulama alanlarından ve farklı karmaşıklık seviyelerinden olması maksadıyla seçilmiştir. Bütün bu arayüzler MODELSIM simülasyon aracı kullanılarak gerçeklenmektedir ve protokolde tanımlanan şekilde mesajları değiştiren gönderici ve alıcı gerçeklenerek test edilmektedir. Buna ek olarak, MIL-STD-1553B protokolü FPGA üzerinde tamamıyla gerçeklenmektedir ve bu uygulamanın doğruluğu ticari bir yonga ile

haberleşilerek kanıtlanmaktadır. Gerçeklemelerin kaynak ve güç tüketim analizleri, mimarinin genelliğinin ve uygulamaların etkinliğinin azaltılmadığını göstermektedir.

UNIBUS, hazır tasarlanmış bloklar ve sinyal arayüzleri sağlayarak, mevcut ve yeni seri veriyolu protokolleri için donanım geliştirme süresini azaltmaktadır. Buna ek olarak, UNIBUS, farklı protokoller arasında ortak olan protokolden bağımsız bileşenler bir kereye mahsus olarak doğrulandığında ve bloklarla birlikte arayüzleri açık bir şekilde tanımlandığında, tasarımın güvenilirliğini artırmaktadır. UNIBUS gerçek sistemlerde kullanılmak üzere tam boyutlu veriyolu arayüzü bileşenlerinin geliştirilmesinde ve varolan ürünlerin testlerinin geliştirilmesinde kullanılabilir. Bu tarz işlemde, verilen veriyolu arayüzünün gerekli fonksiyonları, test edilen bileşen için iletişime geçen karşı parçayı oluşturmak üzere UNIBUS üzerinde uygulanabilir.

Anahtar Kelimeler: Gerçek zamanlı, seri, gömülü veriyolu protokolü, genel mimari, kodlayıcı, kod çözücü, alanda programlanabilir kapı dizileri, donanım, genel veriyolu, seri veriyolu

To My Family

ACKNOWLEDGMENTS

I would like to express my special thanks to my supervisor Assoc. Prof. Dr. Şenan Ece Schmidt. My special thanks go to TÜBİTAK SAGE. Finally, I would like to thank my family and friends who supported me throughout the whole time.

TABLE OF CONTENTS

LIST OF TABLES

TABLES

LIST OF FIGURES

FIGURES

LIST OF ABBREVIATIONS

ARINC	Aeronautical Radio Incorporated
BPC	Bus Protocol Controller
CAN	Controller Area Network
CG	Checksum Generator
CRC	Cyclic Redundancy Check
DBM	Data Buffer Memory
DSP	Digital Signal Processor
FPGA	Field Programmable Gate Array
HDL	Hardware Description Language
LIN	Local Interconnect Network
MOST	Media Oriented Systems Transport
PI	Protocol Independent
PIC	Peripheral Interface Controller
PS	Protocol Specific
TTCAN	Time Triggered Controller Area Network
TTP	Time Triggered Protocol
UART	Universal Asynchronous Receiver Transmitter
UNIBUS	Universal Bus
VHDL	Very High Speed Integrated Circuit Hardware Description Language

CHAPTER 1

INTRODUCTION

Serial communication is widely used in computer networks and embedded systems because of its simplicity and small number of connectors which result in better physical properties such as less crosstalk. Furthermore high data rates are possible with fast clocking.

Examples for serial communication protocols include Ethernet in desktop computing, Controller Area Network (CAN) in in-vehicle networking and manufacturing systems, MIL-STD 1553B in avionic systems, ARINC-708 in Airborne weather radar systems, ARINC-717 in flight data acquisition and recording systems and Universal Asynchronous Receiver/Transmitter (UART) in the communication with peripheral devices.

Serial communication protocols realize the data link layer and the physical layer functionality of the OSI (Open System Interconnection) model [1]. To this end, each protocol defines the physical layer features including the mechanical and electrical characteristics of connectors and communication channel, characteristics of electrical signals and clock synchronization functions. Furthermore, data link layer features are defined including framing, addressing, channel access arbitration, encoding and decoding. Reliable data transmission is achieved by acknowledgement mechanisms, error controlling and handling. One should note that functions related to bit timing and clock synchronization should be carried out with stringent timing.

Despite the high-level similarity, these functions are defined differently for each protocol leading to specific realizations. Recognizing this high-level similarity, this thesis

proposes a modular, generic and abstract hardware architecture UNIBUS (Universal Bus) for implementing serial bus protocols. The architecture supports the timing requirements for bit transmissions and synchronization. The universal designation is because of the high flexibility and configurability of the architecture that enables the realization of different serial bus protocols.

UNIBUS consists of a number of different hardware blocks. All of them differ from each other by the kind of data processing they perform and operate independent of each other. Each block has *protocol independent* (PI) signal interfaces and *protocol specific* (PS) signal interfaces where PS interfaces have to be customized for the specific bus protocol. The design philosophy of UNIBUS is separating the PS and PI functionality and interfaces to increase the re-usability of the PI components and signals as much as possible when implementing different serial bus protocols.

The versatility of UNIBUS is demonstrated by implementing five different serial bus protocols with it. These protocols are MIL-STD-1553B, ARINC-708, ARINC-717, CAN and UART. These protocols are selected to include standards that are similar in terms of functionality and interfaces such as ARINC-708 and ARINC-717 and completely different such as CAN. Also they have different levels of complexity such as the complex MIL-STD-1553B and simple UART.

All of these protocols are simulated by using MODELSIM platform, and then MIL-STD-1553B bus controller is implemented on hardware by using FPGA platform. An experimental set-up with MIL-STD-1553B implemented on UNIBUS and a commercial chip is constructed and the correctness of the implementation is verified by establishing communication between these two parties. We evaluate the hardware resource consumption as well as power consumption for all these implementations. Our results show that despite the architecture is generic the resulting specific realizations are highly efficient. We propose a solution for the decoder design of the hardware framework architecture. Consequently, our proposed decoder design does not require a specific clock signal. Hence, it is possible to design the hardware framework more efficiently and more simply by means of our decoder design.

UNIBUS significantly reduces the workload and the design time of the implementation by reconfiguring the logical organization of the functions of the bus protocols.

The protocol independent components that are common among different protocols need to be verified only once and the protocol specific components are clearly defined. Hence, UNIBUS improves the reliability of the serial bus implementation.

The remainder of the paper is organized as follows. Chapter 2 describes the operation principles of the hardware framework for the serial bus protocols in general and the challenges and performance metrics that we focus on in this thesis during the hardware implementation. The possible hardware implementation platforms and previous works on the implementation of serial bus protocols are also discussed in Chapter 2. We introduce the detailed description of UNIBUS architecture in Chapter 3. Chapter 4 describes in detail the implementation and performance evaluation of the five specific serial bus protocols. Chapter 5 discusses the conclusions and introduces the next stage of our work.

CHAPTER 2

SERIAL BUS STANDARDS AND THEIR IMPLEMENTATION

A real-time system receives data, processes them, and returns the results in a specified time. Real-time bus protocols are intended to serve the communication requirements such real-time systems therefore, they must guarantee response within strict time constraints.

2.1 Serial Bus Standards

Serial bus protocols are the standards that are defined for the bitwise transport of the data on the Serial Bus. Serial bus protocols are used for communication and computer platforms, where the cost of cable and synchronization difficulties make parallel communication impractical.

The functions of the data link layer and the physical layer as defined in OSI layered model are the main functions of the serial bus protocols. Error detection and correction, arbitration, frame generation and frame processing, data stream synchronization and addressing are the functions of the data link layer. Bit decoding, bit encoding, clock synchronization of a bit are the functions of the physical layer. However, each serial bus protocol has own set of algorithm for data processing, the bitwise transmission, and the reception methods.

2.2 Implementation Challenges and Performance Metrics

We identify the following metrics to evaluate UNIBUS both as a generic real time serial bus architecture and for the specific realization of the components.

- Clock frequency requirements (Design of the decoder)
- Structure of the bus architecture (modularity and signal interfaces)
- Logic resource and power consumption
- Memory requirements

Hardware area utilization and hardware power consumption requirements give the performance of our proposed hardware framework. These parameters are used when the simulations of five different bus protocols explained in Chapter 4 are compared.

Memory requirements, and clock frequency requirements are used to compare our implementation of the bus controller terminal of the MIL-STD-1553B bus protocol with the commercial products.

The decoder block for such kind of embedded serial bus protocols continuously monitors its data input lines for a valid character. However, capturing such a valid character is a challenging process. So that, the design of the decoder is important for the complexity of the hardware design. The decoder's method of measuring its two inputs determines its sensitivity to bus line signal changes caused by amplitude variance. Therefore, the transceiver and decoder design must be considered together and the characteristics of the transceiver must be the main guide for the algorithm of the decoding process.

The decoder must estimate the actual zero crossing times while gaps occur between receiver output transitions. The gaps become wider as receive signal amplitude decreases. Decoder must solve this issue in order to decode the received signal correctly.

[65], [66], [60], [58], [18],[21],[28] use specific clock frequency, which is equal to the 12 times of the desired data rate, in their decoder design. They use this clock in order to sample and decode the incoming data. However, in their decoder block

they assume that comparator thresholds are balanced, and zero crossing will be at the midpoint of the input signal. However, this prediction can introduce an additional zero crossing error into the received signal seen by the decoder. Such kind of decoder poorly estimates actual zero crossing times, and it could reject a frame word because of zero crossing errors.

In this thesis, the decoder design in UNIBUS architecture captures the zero crossing points in the encoded data instead of using a specific clock signal. Hence, our decoder design is tolerant of actual zero-cross distortion in the received signal and zero crossing errors are eliminated for our decoder design by means of this method.

Performance evaluation and the comparison results are explained in the Performance Evaluation section of the Chapter 4.

2.3 Related Work on Implementation of Serial Buses

In this Section, we discuss the previous works related to the real-time embedded bus implementation on hardware and specific FPGA implementations.

Works have been done in [41] and [40] to provide a different kind of physical layers for MIL-STD-1553B bus protocol. In [41], new physical layer is analyzed to provide high-speed communication for MIL-STD-1553B protocol whereas in [40] new physical layer is designed to evaluate the channel attenuation of the MIL-STD-1553B protocol system. However, in our work, for implementing such kind of a physical layer integrated circuits are used.

The implementation of UART protocol was presented in [62], [64], [42], and [50]. In [62], three main hardware blocks such as transmitter, receiver, and baud rate generator were designed. In addition, in [64], the analysis of CRC generation algorithm was performed in order to improve the error capability of the design. In [42], a solution for the problems of inefficient data transmission and data bus utilization ratio was proposed whereas in [50], a bidirectional shift converter technique was realized to control package bits of data for data transmission and reception processes. All of these works were implemented on FPGA by using VERILOG hardware description

language.

CAN and UART protocols were studied together in [67]. They used these protocols at the highest transmission rate to investigate the data communication between FPGA and the observer. On the other hand, the design and implementation of UART-SPI Interface were proposed in [38]. SPI slave devices can communicate with a UART port by means of their design.

The implementation of CAN protocol was presented in [46], [39], [37], [56], [45], [61], and [55]. Eight-to-Eleven Modulation technique was proposed for bit-stuffing process to reduce the timing variation caused by bit-stuffing in [46]. A programmable controller was realized in [39]. CAN protocol was used as data communication protocol between the host controller and its various extension modules for their proposed design. In [37], different communication protocols such as LIN, CAN, ByteFlight, TTCAN, FlexRay, MOST, TTP were compared and CAN and LIN protocols were implemented on PIC. In [56], CAN protocol was also implemented on PIC for monitoring parameters such as temperature, battery voltage, and cobalt level in the exhaust. On the other hand, a hardware based CAN sniffer was studied in [45]. The CAN message frame architecture, bit coding techniques and CAN bit sampling techniques were examined in their work. In [61], CAN bus based control system was studied by using FPGA platform whereas in [55], CAN bus based real-time Data Visualization system was studied.

An experimental system for the CAN bus was proposed in [47]. Their design was based on both hardware and software by using micro controller called as STC89C52 and CAN protocol controller called as SJA1000. In addition, in [43], an intelligent controller of charge and discharge machine was implemented. CAN bus communication network was used in their implementation to monitor and control the software of host computer.

A bit-level interference test method was studied in [35], and [54]. Both studies were performed on FPGA. Besides these works, a simulation model for CAN bus was studied in [36]. In [63], a kind of customized dual redundancy CAN bus controller was proposed. Their design was based on FPGA in order to provide high real-time performance and reliability. Physical layer of this design has two independent channels.

If one of them fails during message transmission process, the other one will be used to transmit the message.

The ARINC-659 bus protocol, which is a standard for digital data transfer, was implemented in [34]. Their design was modeled using VERILOG language and their host was modeled by using soft processor core called as NIOS II. HDL modules are interfaced with the soft processor core to achieve the real-time system requirements. On the other hand, in [59], 6-way serial data communication was introduced. Their data processing system was performed on the soft processor called as NIOS II.

2.4 Hardware Implementation Platforms

In this Section, we present the hardware platforms for real-time embedded serial bus architectures concluding that FPGA is the most versatile technology for implementation.

ASIC (Application Specific Integrated Circuit) design yields the fastest implementations for real-time embedded serial bus implementations. However, ASIC design is expensive, and it has a very long time to market and lacks such flexibility.

FPGA (Field Programmable Gate Array) technology is a compromise between the speed of ASIC and the flexibility of the network processors. One of the main reasons for using FPGAs within an embedded system is performance. A very high number of logic gates and embedded blocks in today's FPGAs enables the design of complex hardware platforms with reduced engineering cost and rapid turnaround time. In addition, FPGAs are highly suitable for time critical applications so that they are increasingly employed in safety and mission critical applications within the aerospace and defense sectors. On the other hand, if the final implementation platform is ASIC, FPGA can be used for prototype production [44]. Consequently, it is possible to convert a hardware design on FPGA to ASIC provided that power source design, packaging and boundary scan testing constraints are taken into consideration [51].

Since FPGAs are highly flexible and programmable, the implementation of a real-time embedded serial bus protocols can be done with great ease and flexibility. While

FPGAs are configurable devices, reconfigurable hardware architectures can be easily implemented on FPGAs.

2.5 Previous Implementations of the MIL-STD-1553B protocol on FPGA

Historically, serial communication buses were implemented using commercial off the shelf components. This approach was however not so efficient since the changing application requirements often rendered the system obsolete. Recently, several serial communication protocols are designed to be implemented on the FPGA because more than one communication protocol can be integrated in a single chip and the system can be programmed whenever required by means of the FPGA. However, using FPGAs in such applications has its challenges since time, power, reliability and data integrity are highly crucial factors. Because of that reason, careful designs should be prepared.

There is a large amount of literature on hardware implementation of the serial bus protocols. In this section, we provide an overview of the already existing approaches specifically with respect to MIL-STD-1553B protocol applications, and highlight the non-reconfigurable hardware bus architectures in such kind of systems since we implement MIL-STD-1553B protocol bus controller unit in our implementation. All of the previous studies select FPGA as the target hardware platform, however, the implementations are limited to the VHDL model synthesis without any actual measurements collected on hardware.

A work has been done in developing a specific hardware bus architecture for a serial communication system based on MIL-STD-1553B communication protocol [49] and [52]. However, the authors only state that the MIL-STD-1553B protocol bus controller unit is designed without any further details. Furthermore, no discussion is provided related to the implementation of their designs. The experimental results are provided only for target device utilization in [49] and the simulation result of sending a command word in [52]. Another related work also proposes an implementation of MIL-STD-1553B protocol in an FPGA [65]. In this work, instead of implementing a separate clock, digital phase lock loop is used for data clock recovery,

The work in [66] is also for the implementation of MIL-STD-1553B bus controller

unit. It uses a specific hardware bus architecture for its implementation. It is important to note that their proposed design requires 12 MHz clock frequency for its decoder design since it uses this clock for capturing the encoded data. On the other hand, in [60] VHDL implementation of the Manchester encoder and decoder is purposed. The decoder design for this work also requires a specific clock that has a frequency of 12 times the desired data rate. However, using such kind of specific clock frequency decreases the performance of the system and increases the complexity of the hardware architecture. Therefore, their approach for their decoder design is not preferable for such avionic applications. Another similar work is also proposed in [48]. Encoder and decoder are modeled as a state machine and they are simulated in MODELSIM platform.

A very recent study proposed in [57] suggests a low-cost design of MIL-STD-1553 bus terminal devices. They use DSP(Digital Signal Processor) having on-chip multichannel buffered serial port in order to perform all the necessary functions of MIL-STD-1553B protocol. However, using a DSP for such kind of serial data bus protocols will be slower and more complicated. Using an FPGA for serial bus protocols increases simplicity and the reliability of the system design.

In [58] the authors propose a methodology for studying of MIL-STD-1553 architecture and studying of Core1553BRM IP core from Actel. They implement it on FPGA. Although the title of the paper suggests a VHDL implementation of MIL-STD-1553B protocol, only data transfer between the bus controller unit and remote terminal is demonstrated in this work. For the realization of the MIL-STD-1553B bus protocol, they use IP core so that they didn't design any bus architecture for their implementation.

Specific bus architectures are used for all mentioned previous works. However, in [53] a generic hardware and software architecture for the implementation of serial bus protocols was proposed. The data link layer functions were implemented in software while physical level functions were implemented in hardware. In order that to estimate the efficiency of offered method, a system with 1-Wire interface controller based on UART was implemented. Software and hardware modules were implemented directly in FPGA. Spartan-3E FPGA family [25] was used as a hardware de-

vice whereas soft microcontroller core PicoBlaze [20] provided by Xilinx company for free was used as a soft-core processor.

The efficiency estimation of their UART implementation was made by two criteria: (i) the hardware expenses of system implementation; (ii) time interval required for executing of data exchange. They used a histogram to show their results. Each column in their histogram presents the hardware expenses, the size of application software, the size of interface support software and the amount of used general purpose registers.

Same method is used in order to compare our UART implementation results with this work. Comparison results are explained in the Performance Evaluation section of the Chapter 4.

Furthermore, in most of the proposed works the implementation is limited to the HDL model synthesis without any real-world implementation and test or experiments carried out on hardware. Therefore, based on existing works in the field of avionics, we aim to propose a new generic reconfigurable hardware framework for serial bus communication systems. To the best of our knowledge there is no hardware implementation based on FPGA which uses a generic and reconfigurable hardware architecture like our design.

CHAPTER 3

UNIBUS HARDWARE ARCHITECTURE

In this section, we propose a generic block-level hardware design for the implementation of Real-time Embedded Bus Architectures for serial bus interfaces as depicted in Fig. 3.2 .

We define seven different hardware blocks in UNIBUS architecture according to our functional decomposition of the serial bus interfaces presented in Chapter 1 and Chapter 2 as shown in Fig. 3.2. These hardware blocks are bus protocol controller (BPC), Data Buffer Memory (DBM), Checksum Generator, Encode Controller, Line Encoder, Decode Controller and Line Decoder. These blocks operate independent of each other and have both *protocol independent* (PI) signal interfaces and *protocol specific* (PS) signal interfaces.

The design philosophy of UNIBUS is separating the PS and PI functionality and interfaces to increase the re-usability of the PI components and signals as much as possible when implementing different serial bus protocols.

The high-level functional block diagram and the corresponding hardware block and signal diagram of the UNIBUS architecture are shown in Fig. 3.1 and Fig. 3.2 respectively. The signals with PS functions in Fig. 3.2 are indicated with red color. The width of all multi-bit signals are protocol specific.

Frame transmission and frame reception are two sets of separate processes, interacting with each other. Frame reception consists of the detection of a frame start condition, determining the bit time interval, data stream synchronization, detection of the logical value of the received bit, marking the beginning of a new frame, constructing a frame

out of received bits, address checking, detection and possible correction of frame errors and extraction of the data array (payload) out of a frame, which is read from the data bus.

Frame transmission consists of indication of a frame start condition, determining of the bit time interval, bitwise transmission of the frame, clock synchronization of bits, data stream synchronization, bit stuffing, reception of data array from the application layer and frame generation.

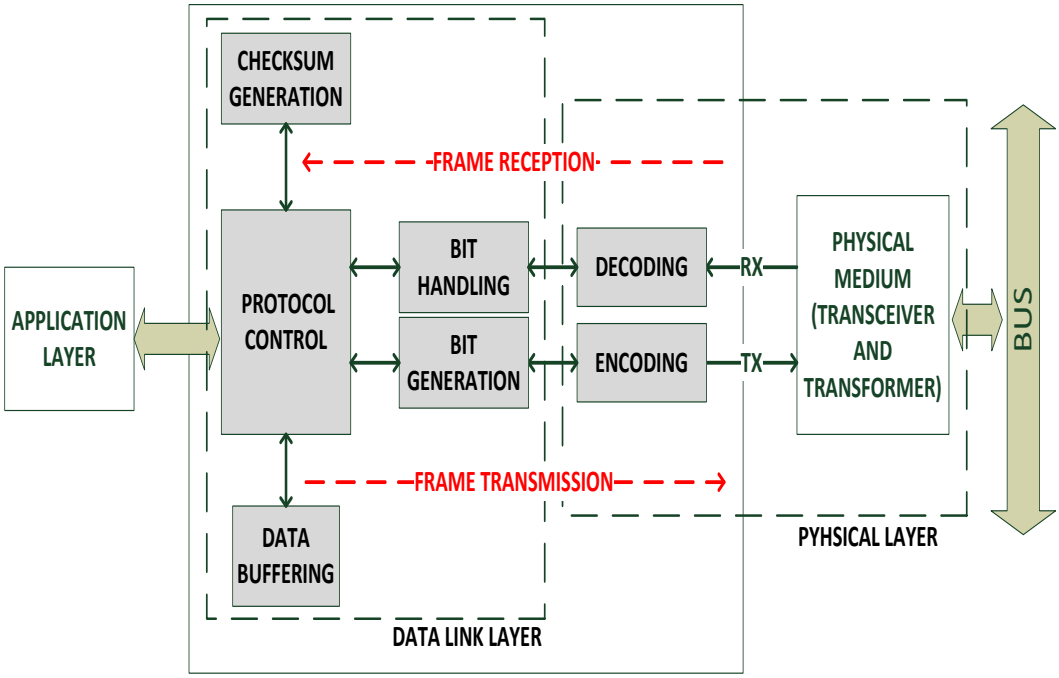

Figure 3.1: The functional block diagram of UNIBUS architecture.

Bus Protocol Controller (BPC) Block is the main block of UNIBUS initiating and controlling the data transfers between UNIBUS and the Application and Physical Layers as well as the data flow among UNIBUS blocks. Frame generation and frame processing, detection and correction of errors in received frames, data stream control, addressing and arbitration are the functions that are implemented in this block. These functions vary according to the type of the bus protocol. Therefore, the logic design of this block should be customized for the implemented protocol.

BPC has input and output signal interfaces with DBM, Checksum Generator, Decoder Controller and Encoder Controller. It communicates with the Checksum Generator

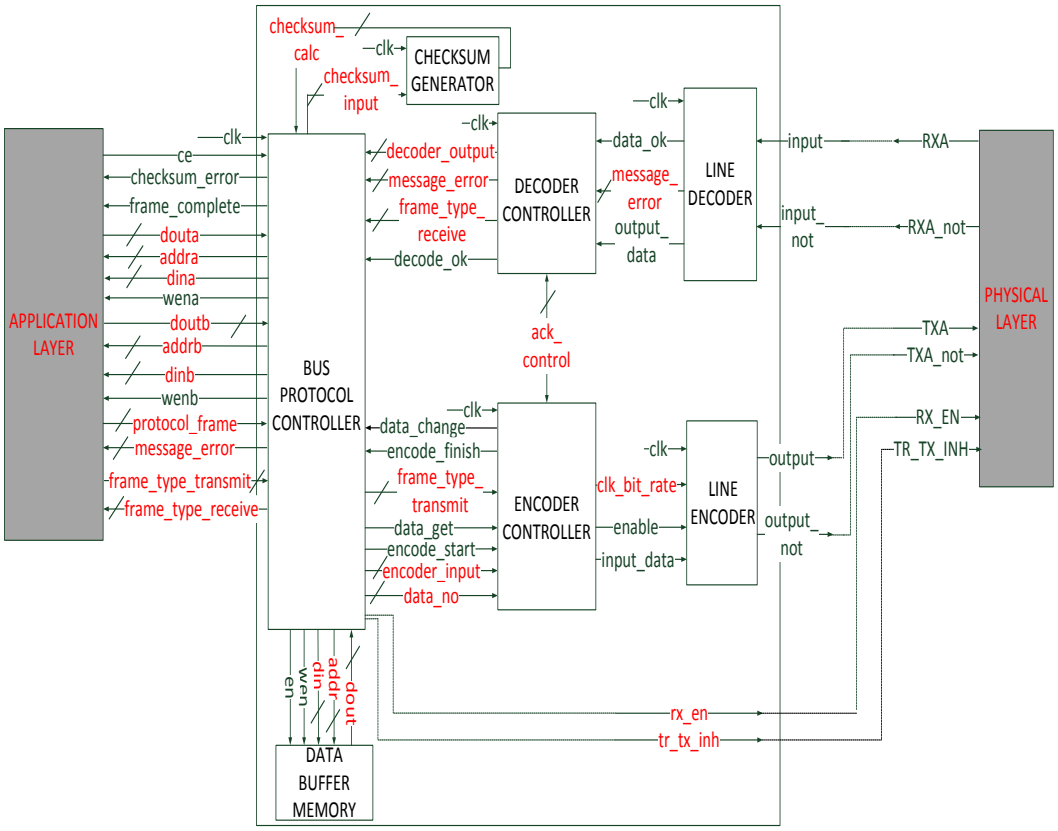

Figure 3.2: Hardware Block Diagram of UNIBUS architecture.

to detect checksum errors, and it communicates with DBM to store the transmitted and received frames. Furthermore BPC realizes frame transmission to the bus and the frame reception from the bus by communicating with the Encoder Controller and Decoder Controller.

BPC has external signal interfaces to application and physical layers. The signal interface between the application layer and BPC is shown in Table 3.1. BPC communicates via a dual port memory with the application layer which enables the users to initiate and monitor the data flow.

BPC sends control signals to the transceiver chip via physical layer signal interface as shown in Table 3.2.

DBM is the data buffer between UNIBUS and physical layer. Its width and depth should be customized for the amount of data that are going to be stored during the frame transmission. BPC communicates with DBM to control the frame transmission

Table 3.1: Signal interface between the Application Layer and BPC.

Signal	Width	Description	Type
ce	1	'1':Enables frame transmission, '0':Disables frame transmission	PI
checksum_error	1	'1': Checksum error, '0':No Checksum error	PI
frame_complete	1	'1':Frame transmission or frame reception is finished, '0':Frame transmission or frame reception continues	PI
douta	PS	Data Output for PORTA of DBM, VALID: Before frame transmission	PI
addra	PS	Address for PORTA of DBM, VALID: Before frame transmission	PI
dina	PS	Data Input for PORTA of DBM, VALID: Before frame transmission	PI
wena	1	'1':Enables writing on PORTA of DBM, '0':Disables writing on PORTA of DBM	PI
doutb	PS	Data Output for PORTB of DBM, VALID: After frame reception	PI
addrb	PS	Address for PORTB of DBM, VALID: After frame reception	PI
dinb	PS	Data Input for PORTB of DBM, VALID: After frame reception	PI
wenb	1	'1':Enables writing on PORTB of DBM, '0':Disables writing on PORTB of DBM	PI
protocol_frame	PS	The frame content, VALID: Before frame transmission and reception	PS
message_error	PS	Protocol related error signals, VALID: After frame transmission and reception	PS
frame_type_transmit	PS	Frame type to be encoded, VALID: Before frame transmission	PS
frame_type_receive	PS	Decoded frame type, VALID: After frame reception	PS

Table 3.2: BPC control signals sent to the transceiver.

Port	Width	Description	Type
rx_en	1	'1':Receiver port of the transceiver is enabled, '0':Receiver port of the transceiver is disabled	PS
tr_tx_inh	1	'1':Transmitter port of the transceiver is disabled, '0':Transmitter port of the transceiver is enabled	PS

and reception via the signal interface shown in Table 3.3.

BPC specifies the address to read with "addr" and reads the data array from DBM on "dout" before frame transmission. In a reverse procedure, after the frame reception, BPC writes the data array extracted from the received frame to DBM by enabling

write with "we". The data are transmitted on "din" interface and stored in the address specified by "addr" signal.

Table 3.3: Signal interface between BPC and DBM.

Port	Width	Description	Type
dout	PS	Data output of DBM, VALID: During frame generation.	PS
addr	PS	Address for DBM, VALID: During frame generation and after frame decoding.	PS
din	PS	Data input of DBM, VALID: After frame decoding.	PS
we	1	'1':Enables writing DBM, '0':Disables writing DBM	PI
en	1	'1':Enables DBM, '0':Disables DBM	PI

Checksum Generator (CG) block generates the checksum value for the frames to be transmitted and as well as the frames that are received. The logic of CG is customized for the implemented protocol to implement different algorithms such as parity error checking or CRC(Cyclic Redundancy Check).

BPC controls and detects the protocol errors. Hence, CG communicates with BPC via the signal interface is shown in Table 3.4. The width of the input for CG depends on the frame length of the implemented protocol.

Table 3.4: Signal interface between BPC and CG.

Port	Width	Description	Type
checksum_calc	PS	Calculated Checksum value, VALID: During frame generation and after frame decoding	PS
checksum_input	PS	Input for Checksum Generator block, VALID: During frame generation and after frame decoding	PS

We partition the encoding of the data to be transmitted into two blocks that are Encoder Controller and Line Encoder to separate the functions and signals which are PI and PS to increase signal reusability.

The Encoder Controller receives the Application data to be encoded in parallel from BPC. It serializes the data, forwards it to the Line Encoder and enables the Line Encoder. The Line Encoder executes the bit transmission functions. It encodes the serial bit stream received from the Encoder Controller in the specific line coding

format such as non-return to zero (NRZ), Manchester bi-phase or Harvard bi-phase. It provides clock synchronization by placing synchronization bits at the beginning of the frame, and it executes bit stuffing if specified.

The design of Decoder Controller and Line Decoder is similar as they reverse the processing of Encoder Controller and Line Encoder. The Decoder Controller converts the serial data received from the Line Decoder to parallel and sends it to BPC. It further establishes the data stream synchronization to enable the decoder to resynchronize at the beginning of each frame received. The Line Decoder executes the bit reception functions. It decodes the serial bit stream received from the data bus in the specific line coding format. The logic design of the blocks is customized for a specific protocol that is implemented.

We next describe UNIBUS operations for the data reception from the Application Layer and frame transmission on the Physical Layer.

The Application layer starts sending data to UNIBUS by setting "ce" signal. "ce" is reset when the transmission is finished. The data from the Application Layer are read on PORTA of the dual port memory by BPC. First, BPC indicates the address of memory to read the data from on "addra" then reads the data on "douta". After reading, BPC writes the data to DBM in order to use them for frame generation.

During frame generation, BPC sends the data to Checksum Generator for the calculation of the checksum value on "checksum_input" and reads the calculated checksum value on "checksum_calc". After this calculation, BPC generates the frame by concatenating data, which are read from the application layer, the checksum value and the necessary frame bits related to the bus protocol.

When frame generation is finished, BPC sets "encode_start" to activate the Encoder Controller. BPC sends the data to the Encoder Controller in blocks. The Encoder Controller reads the number of data blocks to encode on "data_no" together with the first block on "encoder_input". Then, it converts the parallel "encoder_input" signal into a serial stream by shifting it from LSB(least significant bit) to MSB(most significant bit) or vice-versa and sends it to the Line Encoder.

When there are more than one blocks of data transfer, "data_change" and "data_get"

indicate the time of the getting the next data block from BPC. After encoding one block of data, the Encoder Controller sets "data_change" signal to inform BPC for the preparation of the new data block. After that, BPC sets "data_get" signal to inform Encoder Controller for the getting of the new data block. Data blocks are transmitted continuously with no gaps via this handshaking procedure. Therefore, it is a performance requirement that the Encoder Controller ensures no gaps in the encoded data array to support real-time data transmission.

"frame_type_transmit" signal shows the type of the frame that will be encoded. BPC uses this signal for defining the frame type to be generated, and Encoder Controller uses it for defining the synchronization pattern at the beginning of the frame.

The signal interface between BPC and Encoder Controller is shown in Table 3.5.

Table 3.5: Signal interface between BPC and the Encoder Controller.

Port	Width	Description	Type
data_change	1	'1':New data can be sent by BPC, '0': New data can't be sent by BPC	PI
encode_finish	1	'1':Encoding is finished, '0': Encoding is continued	PI
frame_type_transmit	PS	Frame type to be encoded, VALID: During frame encoding	PS
data_get	1	'1':New data can be read by the Encoder Controller, '0':New data can't be read by the Encoder Controller	PI
encode_start	1	'1':Encoding is enabled,'0': Encoding is disabled	PI
encoder_input	PS	Input signal for Encoder Controller, VALID: During frame encoding	PS
data_no	PS	Number of the data array, VALID: During frame encoding	PS

When "encode_start" is set by BPC, the Encoder controller in turn sets "enable" to activate Line Encoder. Signal interface between Encoder Controller and Line Encoder is shown in Table 3.6. The Encoder Controller sends the data serially to the Line Encoder on "input_data". The Line Encoder carries out the bit encoding and sends the encoded bits to the physical layer on "output" and "output_not" signals.

When the last block of data is sent to the Line Encoder on "encoder_input" the

Encoder Controller sets "encode_finish" to indicate the end of the encoding of the frame and resets "enable". "clk_bit_rate" signal is the encoding clock to implement the bit encoding with the proper frequency.

Table 3.6: Signal interface between the Encoder Controller and the Line Encoder.

Port	Width	Description	Type
clk_bit_rate	1	'1':Encoding clock signal is high,'0': Encoding clock signal is low	PS
enable	1	'1':Enables Line Encoder,'0': Disables Line Encoder	PI
input_data	1	Input bit to be encoded, VALID: During frame encoding	PI

Finally, we describe UNIBUS operations for the frame reception from the Physical Layer and data transmission to the Application Layer.

Signal interface between Decoder controller and Line Decoder is presented in Table 3.7.

The "input" is the input bit which is read from the data bus and "input_not" is the complement of it. The Line Decoder continuously monitors "input" and "input_not" starts decoding when it detects a valid character. When the decoding of each data bit is completed, the Line Decoder puts the data bit on "output_data" and sets "data_ok". If the Line Decoder detects a decoding error caused by the input lines, it sets "message_error".

The Decoder Controller reads the decoded data from the "output_data" when the "data_ok" is set. Then, it shifts this decoded bit serially into a shift register. When the last bit is in the shift register, the Decoder Controller sets "decode_ok" to indicate the end of the decoding and puts the decoded data in the shift register on "decoder_output" in order to transmit it to BPC in parallel.

The signal interface between BPC and the Decoder Controller is in Table 3.8. The type of the decoded frame is indicated by "frame_type_receive" signal. It is valid when "decode_ok" signal is set.

BPC writes the data array extracted from "decoder_output" to DBM. After that,

it reads this data array from DBM and writes it on the PORTB of the dual port memory of the application layer.

Firstly, BPC puts the Application Layer Memory address to write the data on "addrb" and sets "wenb" signal to enable writing. Then, it puts the data on "dinb" in order to write them.

Table 3.7: Signal interface between Decoder Controller and the Line Decoder.

Port	Width	Description	Type
data_ok	1	'1': Bit decoding is valid ,'0': Bit decoding is invalid	PI
message_error	PS	Message error signal, VALID: During frame decoding	PS
output_data	1	Decoded output signal, VALID: During frame decoding	PI

Table 3.8: Signal interface between BPC and the Decoder Controller.

Port	Width	Description	Type
decoder_output	PS	Output signal for Decoder Controller, VALID: After frame decoding	PI
message_error	PS	Error signal for Decoder Controller, VALID: After frame decoding	PS
frame_type_receive	PS	Decoded frame type, VALID: After frame decoding	PS
decode_ok	1	'1': Decoding is finished, '0': Decoding is continued	PI

During frame transmission, the transmitter unit can also read the bus to compare whether the sent bit and the read bit are the same or not. Furthermore, during the frame reception, any receiver unit that correctly receives a message can send an acknowledgment bit. In that case, the transmitter unit checks for the presence of this bit at the end of the frame transmission.

Therefore, there is a signal interface between Encoder Controller and Decoder Controller in UNIBUS architecture as shown in Table 3.9. These bit error and acknowledgment bit checking are PS functions.

Table 3.9: Signal interface between the Encoder Controller and the Decoder Controller.

Port	Width	Description	Type
ack_control	PS	Acknowledgment and bit error control signal, VALID: During frame encoding and decoding	PS

CHAPTER 4

IMPLEMENTATION AND EVALUATION OF SELECTED BUS PROTOCOLS WITH UNIBUS

4.1 Design and Implementation

In this Section we implement a number of different serial bus protocols for different applications to demonstrate the use of UNIBUS framework. We choose to implement MIL-STD-1553B bus protocol [3], [12], [19], [6], ARINC-708 bus protocol, ARINC-717 bus protocol [13], [2], CAN bus protocol [26], [5], [10], [27] and UART protocol [33], [14], [15]. Table 4.1 presents a comparative overview of these protocols.

It can be seen in the comparison table that usage areas for these protocols are different. MIL-STD-1553B protocol is used in military applications and ARINC-708 bus protocol is used in radar systems whereas ARINC-717 bus protocol is used in data acquisition and recording systems, CAN bus is used in vehicle applications and UART protocol is the serial communications subsystem of a computer. Consequently, their basic protocol parameters such as bit rate, frame length, data bits per word, message length, message formats, coding type, checksum, synchronization type, operation and terminal are different than each other.

In this section, we provide detailed implementations, which fully demonstrate the features of our proposed generic hardware framework, of these five protocols.

All of the protocols are compiled on FPGA by ISE 14.5 [23] software, and simulated by MODELSIM simulation tool [31]. Furthermore, bus controller terminal of the MIL-STD-1553B bus protocol is fully implemented on a hardware platform.

Table 4.1: Comparison table of the implemented bus protocols.

	MIL-STD-1553B	**ARINC-708**	**ARINC-717**	**CAN**	**UART**
Application	Military specification. Digital Time Division Command/Response Multiplexed Data Bus	Airborne weather radar systems	Flight data acquisition and recording systems	In-vehicle communication, control and automation systems	Serial-parallel conversion with configurable data format and transmission speeds.
Terminal Types	Bus Controller, Remote Terminal, Bus Monitor	Transceiver	Transceiver	Transceiver	Transceiver
Operation	Asynchronous	Asynchronous	Asynchronous	Synchronous	Asynchronous
Baud Rate	1 MHz	1 MHz	768 Hz(nominal rate)	20KHz to 1MHz	Depends on UART model
Frame length	20 bits	1606 bits	768 bits	Max. 108 bits	Max. 12 bits
Data bits per word	16 bits	1600 bits	12 bits	Max. 64 bits	5 to 8 bits
Application Message Length	Max. of 32 data-words + command-word + synchronization bits	1 frame	1 frame (128, 256, and 512 12-bit words per second)	1 frame (data, remote, error, overload)	1 frame
Message Types	- Bus Controller to RT, - RT to bus controller, - RT to RT, - Broadcast, - System Control	- Transmit, - Receive	- Transmit, - Receive	- Transmit, - Receive	- Transmit, - Receive
Coding type	Manchester II bi-phase	Manchester II bi-phase	Harvard bi-phase	Non Return to Zero with bit stuffing	Non Return to Zero
Checksum	Parity	Parity	None	CRC	Parity
Synchronization type	3 synchronization bits	6 synchronization bits	4 synchronization words	Start of frame	Start of frame

UNIBUS is used for both transmitter and receiver models for all protocols. Both data transmission and data reception are simulated for all protocols using separate transmitter and receiver implementations. A bus controller terminal for MIL-STD-1553B is implemented and verified by communicating with a commercial MIL-STD-1553B chip that is used as a remote terminal.

The block diagram for the simulation of MIL-STD-1553B, UART, ARINC-708, and

ARINC-717 bus protocols is shown in Fig. 4.1.

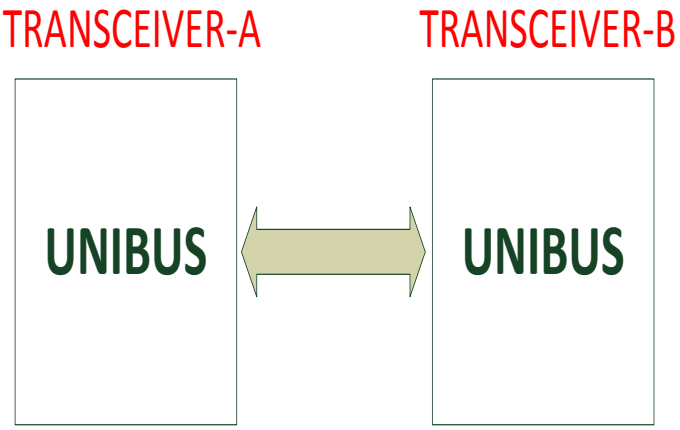

Figure 4.1: Block diagram for the simulation of the MIL-STD-1553B, UART, ARINC-708, and ARINC-717 bus protocols.

The nondestructive bitwise arbitration is used in the CAN protocol. Accordingly, if just one node drives logic low(dominant bit) to the bus, the whole bus is at logic low regardless of the transmitted logic high signals(recessive bit). This process is also realized for our simulation process by using an AND gate between the outputs of the node A and node B. By means of this AND gate logical 0 becomes a dominant bit and logical 1 becomes a recessive bit. The block diagram of the simulation model of the CAN bus protocol is shown in Fig. 4.2.

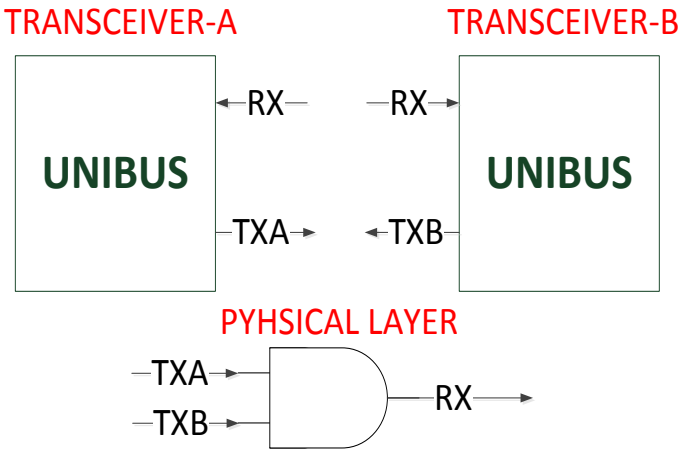

Figure 4.2: Block diagram for the simulation of the CAN bus protocol.

4.1.1 CAN Bus

The CAN bus [26],[5],[10],[27] is a broadcast type of bus and it works as a multi-master system. This means that all nodes can hear all transmissions. CAN protocol defines four different message types which are data frame, remote frame, error frame, and overload frame. Data frame is used to send data information from a source to multiple receivers. Remote frame is used to request the transmission of data information from a remote node. Error frame is used to indicate the detected errors and overload frame is used to request an additional time delay.

We implement two CAN nodes Node A and Node B. These nodes send and receive data frame and remote frames in our simulation. Firstly, node A block sends a remote frame to the node B. After reception process, node B sends corresponding data frame to node A. Nodes listen to the bus as they transmit. Bit error is detected when the bit value that is sent is different from the bit value that is monitored. In addition, acknowledgment error is also detected by a transmitter if a recessive bit (logical 1) is found on the acknowledgment slot of the message frame. CRC error is detected if the CRC computed by the receiver differs from the one stored in the received frame.

According to the CAN protocol, each bit time consists of segments for synchronization. These segments are called Synchronization, Propagation, Phase 1 and Phase 2. Synchronization Segment is used for synchronization of the clocks, and it has one quantum. Propagation Segment compensates the delay in the data bus. Phase 1 segment can be shortened, and Phase 2 segment can be extended to keep the clocks in synchronization. The sample point facilitates continuous synchronization, and it is between Phase 1 segment and Phase 2 segment.

In our simulation model, bit time is divided into ten equal time quanta: one quantum is for synchronization segment, three quanta are for propagation segment, three quanta are for Phase 1 segment and three quanta are for Phase 2 segment. Therefore, receiver samples incoming data bit on the 70% of the bit time.

In this implementation, only two types of frames called as remote frame and data frame were realized during the data transmission and data reception processes. The other defined frames for CAN protocol called as error frame and overload frame will

be implemented as a future work. On the other hand, three types of errors such as CRC error, acknowledgment error, and bit error were checked during the error management process. The other defined errors called as form error and stuff error will be checked as a future work.

4.1.2 UART

UART [33], [14], [15] is used for serial communications with configurable data format and baud rates. It is widely used for communicating to peripheral devices.

Each UART frame begins with a logic low start bit. After the start bit, the data bits (5 to 8 bits, Least Significant Bit (LSB) first) are sent followed by an optional parity bit to be used by the receiver for parity error checking. Lastly, stop bits are sent to indicate the end of transmission.

Because of the asynchronous transmission process of the UART protocol, the transmitter and receiver must agree on protocol parameters (Baud Rate, character length, parity, and stop bits).

According to the UART protocol, a clock signal, which runs at much faster rate than the baud rate, should be used to control the operation of the UART hardware. The receiver looks for the beginning of the start bit by testing the state of the incoming signal. A valid start bit is detected with a high to low voltage transition on the bus line. After waiting for a bit time, the receiver samples the bus line and clocks the result into the shift register.

In [62], [64], [38],and [50] a frequency equal to 16 times of the baud rate is used as a reference clock for their UART designs. However, in our simulation model, we choose the baud rate as 1 Mbps and the UART module has an internal clock (100 MHz) that runs 100 times faster than the baud rate. By means of this clock, the UART receiver samples the incoming data with the sampling rate of 100. This causes greater immunity against baud rate error.

UART protocol works as a transceiver. Therefore, We implement two UART nodes Node A and Node B as transceivers and test transmit and receive messages in the

simulation.

Firstly, message is sent by Node A and then it is received by Node B terminal. After that, received message is sent by Node B and then it is received by Node A.

In this implementation, UART protocol was completely simulated by implementing both data transmission and data reception processes. In addition, parity error checking process was also realized for both processes. A hardware implementation of this protocol will be realized as a future work.

4.1.3 ARINC-708 and ARINC-717

Only data frame is defined for both ARINC-708 and ARINC-717 [13],[2] protocols. Data frame is 1600 bits long and baud rate is 1 MHz for ARINC-708. Each frame starts with a 3 us sync pattern (1.5 us high, 1.5 us low) and ends with another 3 us sync pattern (1.5 us low, 1.5 us high). Therefore, a complete frame takes 1606 us. On the other hand, data frame is 768 bits long and bit rate is 768 Bps for ARINC-717. This consists of 64 words(12 bits length). Complete frame has four subframes (one subframe per second) because of that reason it takes 4 s. The first word in each subframe provides a synchronization pattern for this protocol.

These bus protocols also work as a transceiver. It means that, terminals can either transmit or receive messages. Because of that reason, during the simulation process both transmit and receive messages are tested. Firstly, message is sent by transceiver A terminal and then it is received by the transceiver B terminal. After that, received message is sent by transceiver B terminal and then it is received by the transceiver A terminal.

In this implementation, ARINC-708 and ARINC-717 protocols were completely simulated by implementing both data transmission and data reception processes. In addition, parity error checking process was also realized for both protocols. A hardware implementation of these protocol will be realized as a future work.

4.1.4 MIL-STD-1553B

In the scope of this thesis, we fully implement MIL-STD-1553B protocol using UNIBUS architecture. To this end the hardware design is carried out according to the UNIBUS block definitions and signal interfaces. Then the design is implemented on Xilinx SPARTAN3-XC3S4000-4FG900 [25] FPGA device.

We select MIL-STD-1553B for full implementation as it is both a widely used and complicated protocol with many different message types.

MIL-STD-1553B is one of the oldest and the most common serial data bus used in avionic systems. It is used in various military, avionics and aerospace systems for last four decades because of its robust performance, high level of reliability and fault tolerance in harsh environments.

There are two node types for MIL-STD-1553B which are called bus controller and remote terminal (RT) [3], [12],[19],[6]. The bus controller works as a master and issues commands, which are for the transfer of data or the control and management of the bus, onto the data bus. The remote terminal works as a slave. We implement the bus controller using UNIBUS.

There are two types of message transfer formats which are Information transfer format and broadcast information transfer format. Six message types which are bus controller to RT transfer, RT to bus controller transfer, RT to RT transfer, mode command without data word, mode command with data word (transmit) and mode command with data word (receive) are defined for information transfer format. Four message types such as bus controller to RT(s) broadcast, RT to RT(s) broadcast, mode command without data word and mode command with data word (receive) are defined for the broadcast information transfer format.

For all message types, firstly the bus controller sends a command word to the bus to indicate the data flow and then corresponding remote terminal responds this command word by sending status or data words. If the command word sent by bus controller corresponds to the transmit command, bus controller block sends data words to the remote terminal. After that, remote terminal sends status word to the bus controller.

If the command word sent by bus controller corresponds to the receive command, remote terminal sends data and status words. Because of that reason, both bus controller and remote terminal blocks are designed as transceiver blocks in our simulation.

In this implementation, MIL-STD-1553B bus protocol was completely simulated. All of the defined message types and error checking processes were realized during the data transmission and data reception processes. Detailed error management process for the validation test of this protocol will be implemented as a future work.

4.1.5 Summary and Overview of Signal Interfaces

Serial bus protocols have similar functionality such as bit encoding, bit decoding, error detection, addressing, arbitration, data stream control and frame generation. However, the content of these functions differ from one protocol to another. Therefore, PS signal interface is used between the sub-blocks of the UNIBUS architecture to provide generic signal interface for all serial bus protocols.

In this Section, we demonstrate the versatility of UNIBUS in Table 4.2, Table 4.3 and Table 4.4 by listing the signal interfaces of the different serial protocols that we implement. We indicate the PS signal widths as well as their functions.

4.1.6 MIL-STD-1553B Hardware Realization

The simulated model for the MIL-STD-1553B bus controller is further implemented on FPGA. We present the hardware block diagram of the implementation and the test bench in Fig. 4.3.

The implementation platform contains an FPGA, a Digital Signal Processor (DSP), a PROM to keep the FPGA code, a flash memory to store non-volatile data for DSP, a remote terminal chip for MIL-STD-1553B bus protocol, an ethernet interface to provide a monitor interface, physical layers for MIL-STD-1553B bus and ethernet port and power converters to produce the different voltage levels required by the FPGA and the DSP.

Table 4.2: The detailed PS signal interface between the Application layer and BPC block for the implemented serial protocols.

UNIBUS	**CAN**	**UART**	**ARINC-708**	**ARINC-717**	**MIL-STD-1553B**
-	**Width / Content**	**Width / Content**	**Width / Content**	**Width / Content**	**Width / Content**
douta	16 / Data to be transmitted	8 / Data to be transmitted	16 / Data to be transmitted	12 / Data to be transmitted	16 / Data to be transmitted
dina	16 / not used	8 / not used	16 / not used	12 / not used	16 / not used
doutb	16 / not used	8 / not used	16 / not used	12 / not used	16 / not used
dinb	16 / Data read from the bus	8 / Data read from the bus	16 / Data read from the bus	12 / Data read from the bus	16 / Data read from the bus
protocol_frame	15 / Arbitration field, data length code	3 / 1 parity bit; 2 stop bits	60 / bits for header field	48 / sync frames	16 / terminal address, transmit receive bit, subaddress mode, wordcount modecode
message_error	3 / ACK, CRC, decoder error	2 / parity error, framing error	3 / header parity error, data parity error, decoder error	2 / parity error, decoder error	15 / errors for status word, status parity error, data parity error, parity error, nonresponsetimeout error, decoder error
frame_type_transmit	2 / data, remote, error, overload frame	1 / not used	1 / not used	1 / not used	2 / bus controller to rt, rt to bus controller, rt to rt, mode commands
frame_type_receive	2 / data, remote, error, overload frame	1 / not used	1 / not used	1 / not used	2 / bus controller to rt, rt to bus controller, rt to rt, mode commands

In hardware implementation, transmit and receive message types are realized in general hence there have to be a bus controller and a remote terminal. Our proposed hardware framework is used as a bus controller terminal while a commercial product called as HI-6121 [30] is used as a remote terminal. Users can monitor the message flows happened on the data bus, by using graphical user interface. This graphical user

Table 4.3: The detailed PS signal interface between the BPC, DBM, CG, Encoder Controller and Decoder Controller blocks for the implemented serial protocols.

UNIBUS	**CAN**	**UART**	**ARINC-708**	**ARINC-717**	**MIL-STD-1553B**
-	**Width / Content**	**Width / Content**	**Width / Content**	**Width / Content**	**Width / Content**
dout	16 / Data to be transmitted	8 / Data to be transmitted	16 / Data to be transmitted	12 / Data to be transmitted	16 / Data to be transmitted
din	16 / Data read from the bus	8 / Data read from the bus	16 / Data read from the bus	12 / Data read from the bus	16 / Data read from the bus
checksum_calc	16 / Checksum Value	1 / Checksum Value	1 / Checksum Value	1 / Checksum Value	1 / Checksum Value
checksum_input	19 / the stream of bits from the START OF FRAME bit to the DATA FIELD	8 / the stream of bits of the DATA FIELD	16 / the stream of bits of the DATA FIELD	12 / the stream of bits of the DATA FIELD	16 / the stream of bits of the DATA, COMMAND, and STATUS WORDS
frame_type_transmit	2 / data, remote, error, overload frame	1 / not used	1 / not used	1 / not used	2 / bus controller to rt, rt to bus controller, rt to rt, mode commands
encoder_input	44 to 108 / All frame	11 / All frame	16 / the stream of 16 bits of the DATA FIELD	12 / the stream of 12 bits of the DATA FIELD	17 / All frame without sync bits
data_no	3 / the number of the data	1 / the number of the data	7 / the number of the data	7 / the number of the data	5 / the number of the data
message_error	2 / ACK error, decoder error	1 / framing error	1 / decoder error	1 / decoder error	1 / decoder error
decoder_output	44 to 108 / All frame	11 / All frame	16 / the stream of 16 bits of the DATA FIELD	12 / the stream of 12 bits of the DATA FIELD	17 / All frame without sync bits
frame_type_receive	2 / data, remote, error, overload frame	1 / not used	1 / not used	1 / not used	2 / bus controller to rt, rt to bus controller, rt to rt, mode commands

interface communicates with the hardware by using ethernet protocol. For this purpose, in our hardware platform DP83848I ethernet physical layer [9] and HX1188NL transformer [8] are used as a physical layer for the ethernet terminal.

Table 4.4: The detailed PS signal interface between the Decoder Controller and Encoder Controller blocks and the Decoder Controller and Line Decoder blocks for the implemented serial protocols.

UNIBUS	**CAN**	**UART**	**ARINC-708**	**ARINC-717**	**MIL-STD-1553B**
-	**Width / Content**	**Width / Content**	**Width / Content**	**Width / Content**	**Width / Content**
message_error	1 / decoder error	1 / framing error	1 / decoder error	1 / decoder error	1 / decoder error
ack_control	45 to 109 / Contains All frame and ACK_send = '1' :receiver can send dominant ACK bit; = '0' :receiver can not send dominant ACK bit;	- / not used	- / not used	- / not used	- / not used

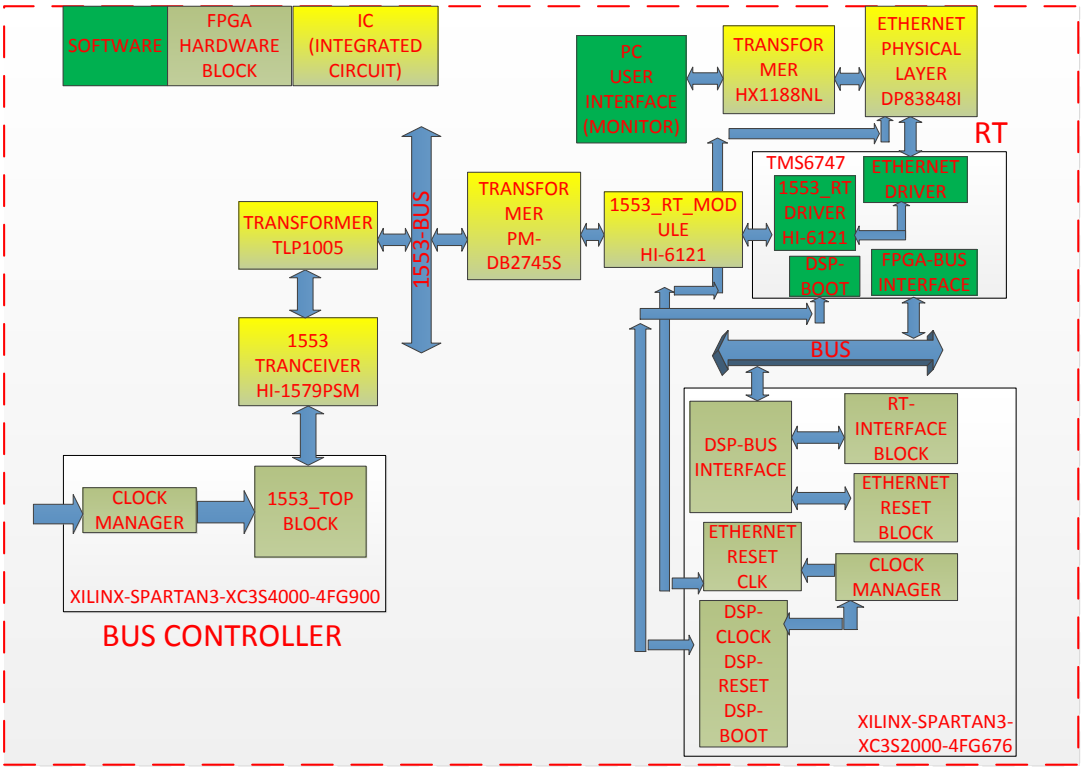

Figure 4.3: Block diagram of the hardware platform for the implementation of bus controller terminal of the MIL-STD-1553B protocol.

Any hardware devices that satisfy the requirements for our proposed design can be used in this implementation. We choose Xilinx SPARTAN3-XC3S4000-4FG900 [25] FPGA device to realize our proposed hardware architecture. XCF32PVO48C PROM chip [16] is used to keep the FPGA code. On the other hand, Xilinx SPARTAN3-XC3S2000-4FG676[25] FPGA device and Digital Signal Processor chip called as TMS320C6747 [32] are used in the remote terminal block of the implementation platform. XCF08PVOG48C PROM chip [16] is used to keep the FPGA code. FPGA device provides necessary clock and control signals for the other devices used in the remote terminal block. In addition, it is also used to boot the DSP device. DSP chip is used to monitor the data bus by using ethernet protocol driver. In addition, HI-6121 chip is driven by this processor in order to communicate with this chip as a remote terminal of the MIL-STD-1553B bus protocol. A flash memory called as AT25DF161-SH-B [22] is used to store non-volatile DSP data. Since both FPGA and DSP chips are used in the remote terminal platform, they have to communicate to each other. Because of that reason, communication drivers are also realized on hardware and software for both devices.

HI-1579PSM MIL-STD-1553B transceiver chip [29] and TLP-1005 transformer chip [4] are used as a physical layer for the bus controller terminal, and PM-DB2745S transformer chip [24] is used as a physical layer for the remote terminal. Transceiver chip for the remote terminal is embedded to HI-6121 chip.

Transceiver is used to remove high-frequency distortion and to add as little error as possible to the input waveform. The bus receiver of the HI-1579PSM chip contains two voltage comparators that provide a pair of digital outputs to the decoder [7]. One comparator has a negative threshold and thus detects negative excursions of the input signal from the bus. The other one has a positive threshold and detects positive excursions.

Transitions in outputs of the comparators occur at the different times from the actual zero crossing points which should be at the mid-point of the bit time. One comparator switches a little earlier than the zero crossing point and the other switches a little later. Ideally, the zero crossing point must happen halfway between the transitions of the two comparators according to the Manchester coding technique which is a main

coding style of the MIL-STD-1553B bus protocol. However, unbalanced comparator thresholds modify this zero crossing point.

Screen shot of the scope screen that was taken from the output signals of the transceiver called as HI-1579PSM is shown in the Fig. 4.4. In this figure, yellow graph corresponds to the positive output, the green graph corresponds to the negative output and purple graph corresponds to the difference between them.

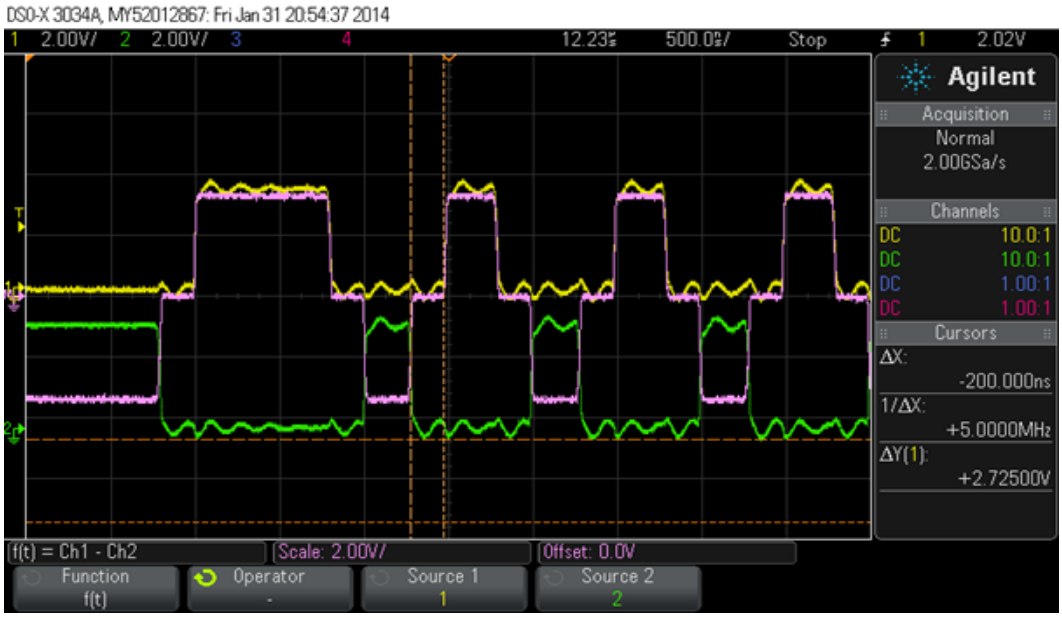

Figure 4.4: Screen shot of the scope screen. Outputs of the transceiver (HI-1579PSM).

For applications, where a transceiver interfaces digital devices, the logic designers must realize that decoder's method of measuring its two inputs to estimate actual zero crossing times determines its sensitivity to bus line signal changes caused by amplitude variance. Thus, the transceiver and decoder designs must be considered as a whole. Therefore, characteristics of the transceiver used in the overall design of the MIL-STD-1553B system must be the main guide for the algorithm of the decoding process. In our implementation, HI-1579PSM transceiver chip is used. Hence, the timing characteristics of this transceiver specifies the algorithm of our designed decoder block.

According to the input waveform compatibility section of the MIL-STD-1553B bus protocol, the terminal shall receive and operate with the incoming signals varied from the square wave to the sine wave. In addition, its tolerance for zero crossing deviations must be $\pm$150 ns. It means that, decoder must decode incoming signals that have up to $\pm$150 ns of error in any time interval between zero crossings.

The rise and fall time requirements in the signal transmission of the MIL-STD-1553B bus protocol cause gaps between outputs of the comparators. The decoder must estimate the actual zero crossing times while gaps occur between receiver output transitions. The gaps become wider as receive signal amplitude decreases. Decoder must solve this issue in order to decode the received signal correctly. Our decoder design uses the zero crossing points to decode the incoming data. Hence, gaps between signal transitions are not important for our decoder block.

Previous works mentioned in [18],[21],[28] use specific clock frequency, which is equal to the 12 times of the desired data rate, in their decoder design. They use this clock in order to sample and decode the incoming data. However, in their decoder block they assume that comparator thresholds are balanced and zero crossing will be at the midpoint of the input signal but this prediction can introduce an additional zero crossing error into the received signal seen by the decoder. Such kind of decoder poorly estimates actual zero crossing times, and it could reject an MIL-STD-1553B word because of zero crossing errors. Therefore, we capture the zero crossing points in the encoded data instead of using a specific clock signal. Hence, our decoder design is tolerant of actual zero-cross distortions in the received signal and zero crossing errors are eliminated for our decoder design by means of this method.

As it is seen in Fig. 4.3 that there are two main hardware blocks called as CLOCK MANAGER and 1553_TOP in the bus controller block. Detailed block diagram of it is seen in Fig. 4.5. DCM CLOCK GEN, CLOCK BUFFER, and CS COUNTER blocks are the sub-blocks for the clock manager block while 1553 TOP MODULE is the sub-block for the 1553_TOP block. Detailed signal interfaces of these blocks are explained in the table 4.5.

"clk_25MHz" signal is the input oscillator clock signal. This signal is used by DCM CLOCK GEN block to produce "clk_100MHz" signal. On the other hand, DCM

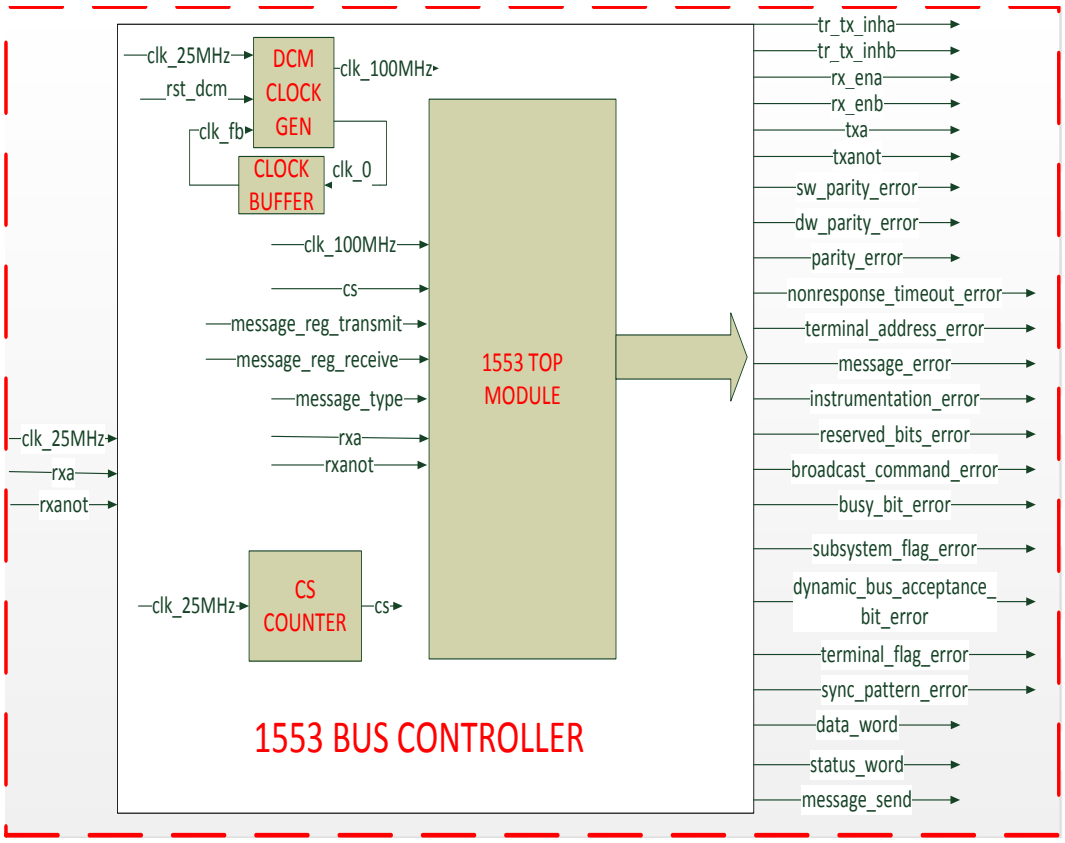

Figure 4.5: Block diagram for the bus controller block of the MIL-STD-1553 protocol.

CLOCK GEN block needs a feedback and a reset signal in order to produce a clock signal. "`rst_dcm`" is the reset signal and "`clk_fb`" is the feedback signal of the this block. CLOCK BUFFER block is used to provide this feedback. For this purpose, it uses "`clk_0`" signal. "`cs`" is chip select signal for 1553 TOP MODULE block, and CS COUNTER block produces it. "`rxa`" and "`rxanot`" are the input signals that are received from the bus. "`message_type`" signal indicates the type of the message whereas "`message_reg_transmit`" and "`message_reg_receive`" signals indicate the command words that are going to be served by the bus controller.

"`txa`" and "`txanot`" signals are the encoded signals sent to the bus. "`tr_tx_inhb`", "`tr_tx_inhb`", "`rx_ena`" and "`rx_enb`" are the control signals for the transceiver chip. "`message_send`" signal indicates the end of the message transmission.

"`data_word`" signal is the received data word while "`status_word`" signal is

Table 4.5: Signal interface of the MIL-STD-1553 bus controller block.

Port	Width	Description
clk_25MHz	1	25 MHz clock signal
rxa	1	Input data signal
rxanot	1	Complement of the input data signal
rst_dcm	1	Reset signal for digital clock manager
clk_0	1	Input signal for clock buffer
clk_fb	1	Feedback signal for the DCM block
clk_100MHz	1	100 MHz clock signal
cs	1	Chip select signal
message_reg_transmit	16	Transmit command word message
message_reg_receive	16	Receive command word message
message_type	4	Message type control signal
tr_tx_inha	1	Transmit inhibit for BUSA
tr_tx_inhb	1	Transmit inhibit for BUSB
rx_ena	1	Receive enable for BUSA
rx_enb	1	Receive enable for BUSB
txa	1	Encoded output signal
txanot	1	Complement of the encoded output signal
sw_parity_error	1	Status word parity error
dw_parity_error	1	Data word parity error
parity_error	1	Parity error
nonresponse_timeout_error	1	Nonresponse timeout error
terminal_address_error	1	Terminal address error
message_error	1	Message error
instrumentation_error	1	Instrumentation error
reserved_bits_error	1	Reserved bits error
broadcast_command_error	1	Broadcast bit error
busy_bit_error	1	Busy bit error
subsystem_flag_error	1	Subsystem flag error
dynamic_bus_acceptance_bit_error	1	Dynamic bus acceptance error
terminal_flag_error	1	Terminal flag error
sync_pattern_error	1	Synchronization pattern error
data_word	16	Data word signal
status_word	16	Status word signal
message_send	1	Message send indication signal

the received status word from the bus. "parity_error" signal is the parity error bit happened on data flow on the bus. "dw_parity_error" signal is the parity error bit for the data word whereas "sw_parity_error" is the parity error bit for the status word. "sync_pattern_error" signal points to synchronization pattern error. According to the MIL-STD-1553B bus protocol, the sync pattern is a waveform with six 0.5 us divisions. Each division is represented as a 1 or 0 to indicate the polarity of the divisions on the data bus. A proper status sync is symbolized as

111000, and a proper data sync is symbolized as 000111. If decoder block captures different sync patterns, it sets the "sync_pattern_error" signal.

According to the MIL-STD-1553B bus protocol, remote terminal shall respond with a valid status or the number of valid data words requested, at time equal to 14.0 us after the valid command word. If the remote terminal does not respond the command word within 14.0 us, "nonresponse_timeout_error" signal is produced.

The other error signals that are generated by 1553 TOP MODULE block are produced by using the status word send by the remote terminal. If the remote terminal detects any error related to the data flow happened on the bus, it sets the corresponding error bit on the status word and send it to the bus controller. The subsystem flag bit in the status word is used by the RT to alert the BC that a subsystem fault exists while the terminal flag bit in the status word indicates a fault in the RT itself. In addition, the message error bit is used by the RT to indicate an error in the command or data words. "subsystem_flag_error", "terminal_flag_error" and "message_error" are the error signals that are send by the remote terminal.

On the other hand, bus controller verifies RT address bits, instrumentation bit and reserved bits of the status word. If the RT address bits do not match the RT address send by bus controller, "terminal_address_error" signal is produced. In addition, if the instrumentation bit and reserved bits do not match their expected values, "instrumentation_error" and "reserved_bits_error" signals are produced.

The bus controller can use the broadcast command received bit in the status word to determine whether a broadcast command was received error-free or not. If there is a error for this bit, "broadcast_command_error" signal is set.

"busy_bit_error" and "dynamic_bus_acceptance_bit_error" are produced if their corresponding bits in the status word indicate an error condition.

According to the MIL-STD-1553B bus protocol, there are ten types of messages as it is mentioned before. 1553 TOP MODULE block supports all message types. A detailed block diagram of this block is seen in Fig. 4.6 and its signal interface is explained in table 4.6. It can be seen on the figure that, each message type has their

hardware blocks. These blocks and their "`input`" and "`output`" signals correspond to our proposed hardware architecture. Multiplexer is used to enable one of these blocks and demultiplexer is used to choose corresponding output signals according to the message type that is served by the bus controller.

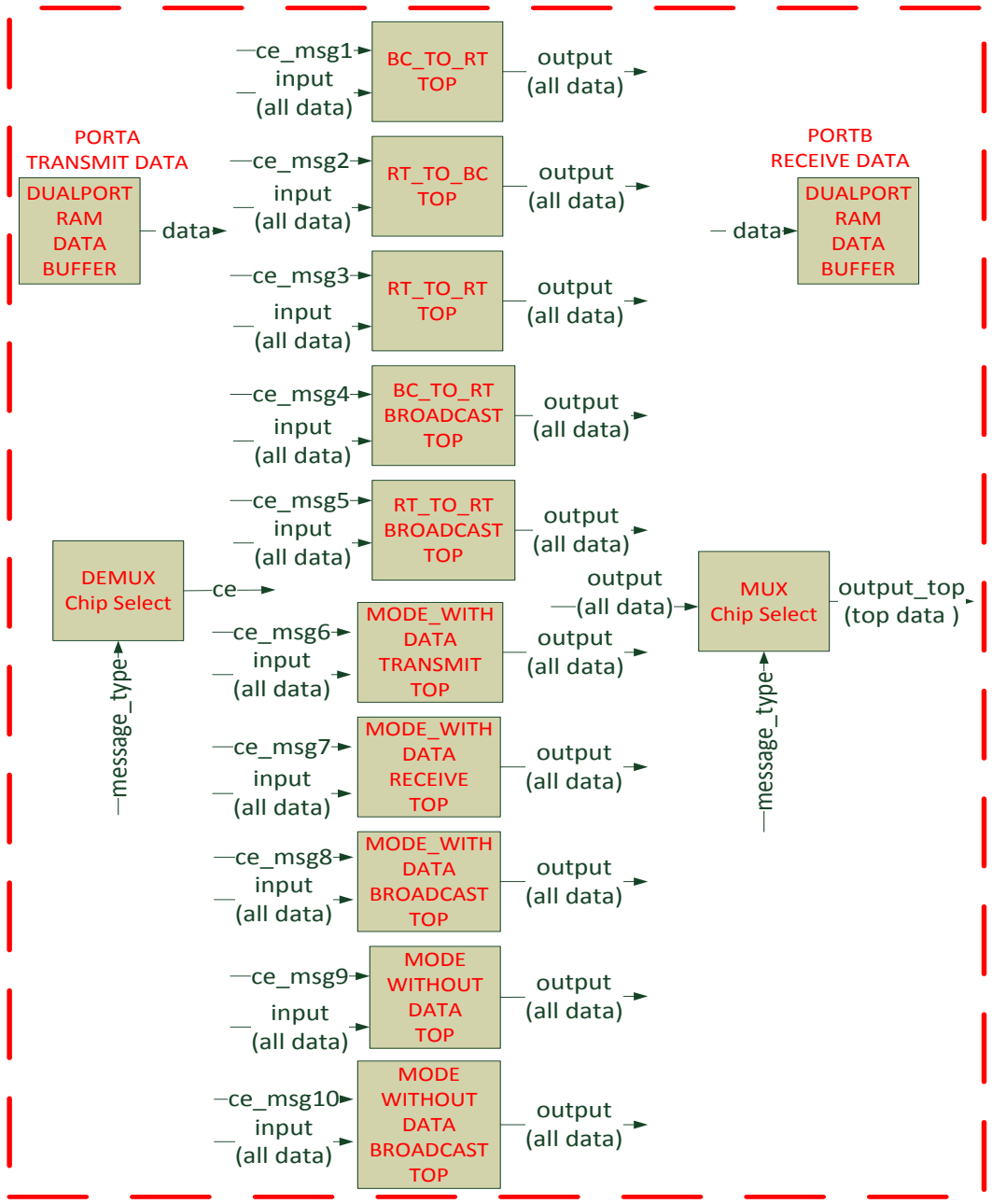

Figure 4.6: Block diagram for 1553 TOP MODULE block.

On the other hand, dual port memory block is used as a data buffer between the application layer and the data link layer. A detailed block diagram of this memory block is

Table 4.6: Signal interface of the 1553 TOP MODULE block.

Port	Width	Description
message_type	4	Message type control signal
ce	1	Chip enable signal
ce_msg1	1	Chip enable signal for message type1
ce_msg2	1	Chip enable signal for message type2
ce_msg3	1	Chip enable signal for message type3
ce_msg4	1	Chip enable signal for message type4
ce_msg5	1	Chip enable signal for message type5
ce_msg6	1	Chip enable signal for message type6
ce_msg7	1	Chip enable signal for message type7
ce_msg8	1	Chip enable signal for message type8
ce_msg9	1	Chip enable signal for message type9
ce_msg10	1	Chip enable signal for message type10
data	16	Data word signal of the data buffer memory
input	-	All input signals that are described for UNIBUS architecture
output	-	All output signals that are described for UNIBUS architecture
output_top	-	Output signals of the top block

attached in Fig. 4.7 and its signal interface is explained in table 4.7. Firstly, transmitted data are written to the Port A of this memory by the application layer. If the bus controller serves a transmit message type, desired data are read from this port of the data buffer. On the other side, if the bus controller serves a receive message type, received data are written to the Port B of this memory during the message transmission process by the data link layer. Then, these are read from this port by the application layer in order to monitor them. Cycle diagram of these data buffer is shown in Fig. 4.8.

Apart from this dual port memory, a single port memory is used in our proposed hardware architecture as it is seen in Fig. 3.2. This memory is used as a data buffer between the data link layer and the physical layer. A detailed block diagram of this memory block is attached in Fig. 4.9 and its signal interface is explained in table 4.8. If the bus controller serves a transmit message type, desired data are read from this buffer. In addition, if the bus controller serves a receive message type, received data are stored to this buffer. Cycle diagram of this data buffer is shown in Fig. 4.10.

A loopback test is realized for testing the functionality of our design. Result of such

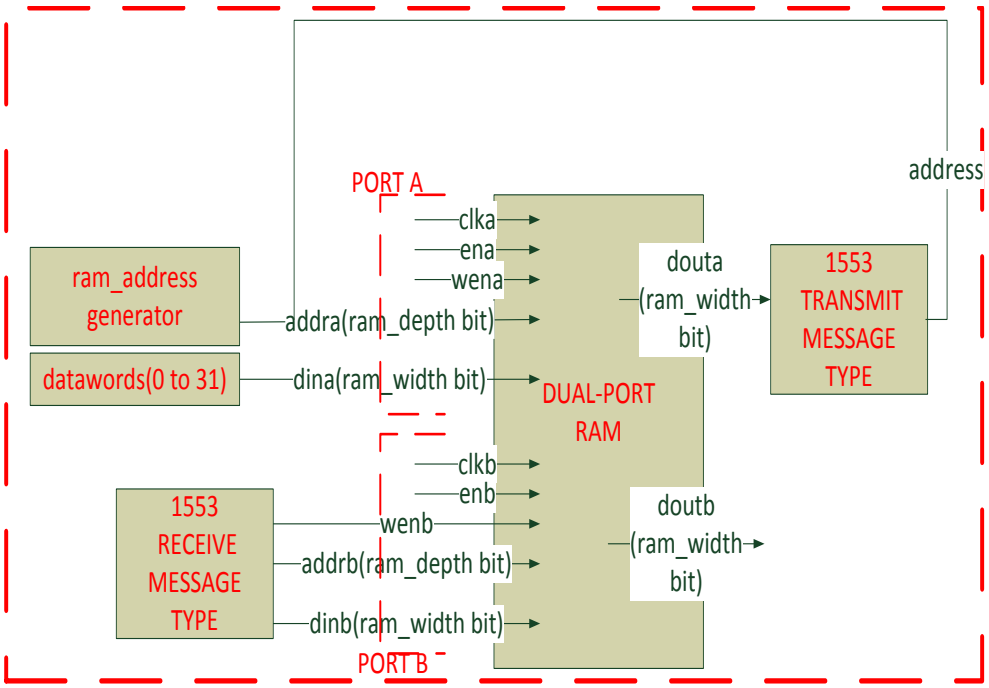

Figure 4.7: Block diagram for the data buffer between the application layer and the data link layer.

Table 4.7: Signal interface of the dual port memory block.

Port	**Width**	**Description**
clka	1	Clock signal of portA of the memory
ena	1	Enable signal of portA of the memory
wena	1	Write enable signal of portA of the memory
addra	ram depth bits	Address signal of portA of the memory
dina	ram width bits	Input data of portA of the memory
clkb	1	Clock signal of portB of the memory
enb	1	Enable signal of portB of the memory
wenb	1	Write enable signal of portB of the memory
addrb	ram depth bits	Address signal of portB of the memory
dinb	ram width bits	Input data of portB of the memory
douta	ram width bits	Output data of portA of the memory
doutb	ram width bits	Output data of portB of the memory
address	ram depth bits	RAM address signal of the transmit message block

kind of loopback test determines whether the both data transmission and data reception processes works well or not. Two different message types called as bus controller

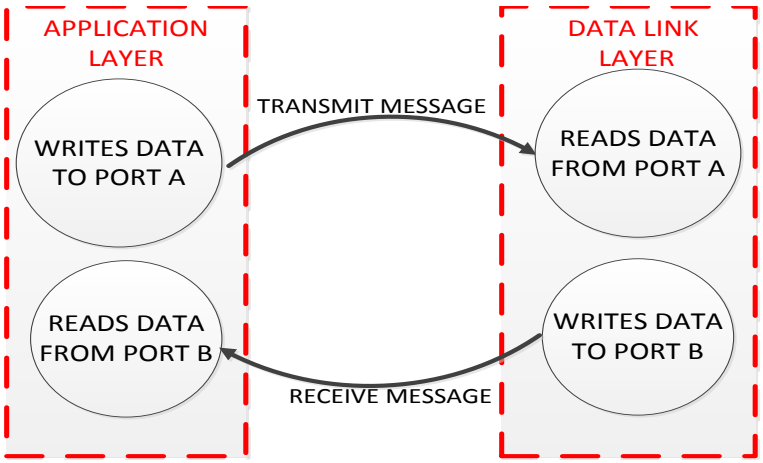

Figure 4.8: Cycle diagram for the data buffer between the application layer and the data link layer.

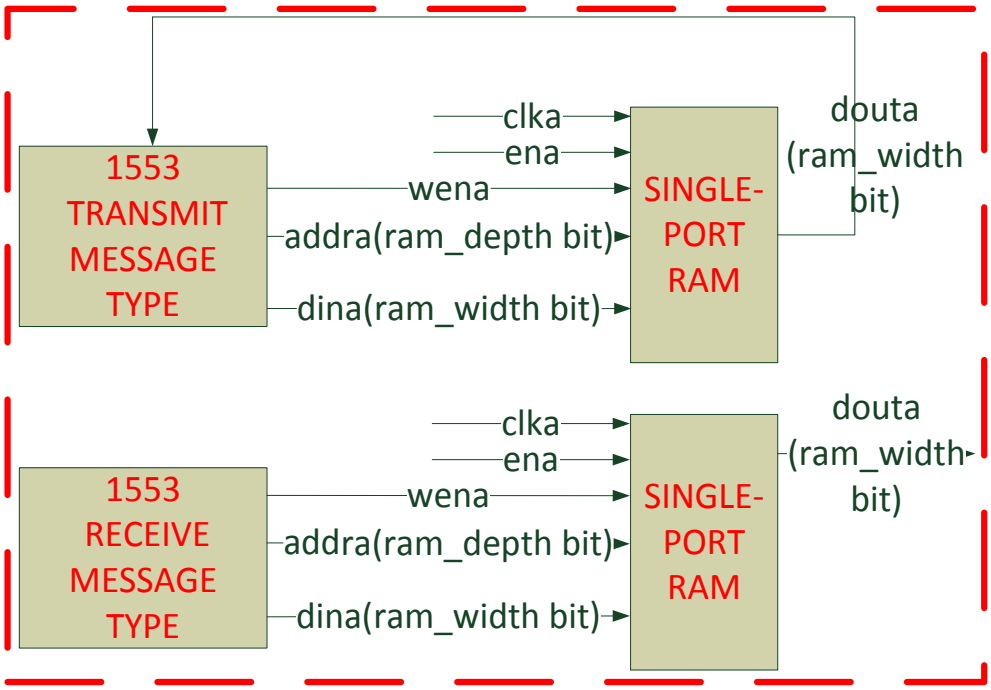

Figure 4.9: Block diagram for the data buffer between the data link layer and the physical layer.

to RT transfer (transmit) and RT to bus controller transfer (receive) are used for this purpose. Firstly, bus controller terminal sends a receive command word (RT to BC) to request three data words from a remote terminal (HI-6121). A screen shot of this command word is shown in Fig. 4.11. After a valid command, remote terminal (HI-

Table 4.8: Signal interface of a single port memory block.

Port	Width	Description
clka	1	Clock signal of the memory
ena	1	Enable signal of the memory
wena	1	Write enable signal of the memory
addra	ram depth bits	Address signal of the memory
dina	ram width bits	Input data of the memory
douta	ram width bits	Output data of the memory

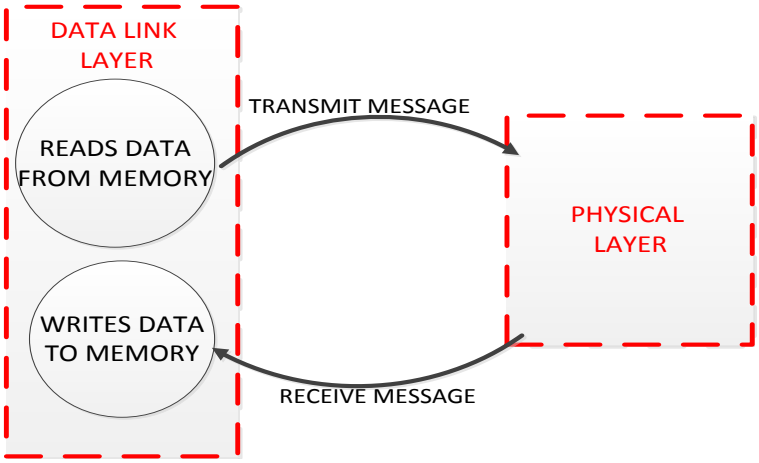

Figure 4.10: Cycle diagram for the data buffer between the data link layer and the physical layer.

6121) sends status word and desired number of data words to the bus controller and monitor these data on the user interface which is shown in Fig. 4.14. The screen shot of these status and data words are shown in Fig. 4.12. After reception of valid status and data words, bus controller terminal sends these received data words to a remote terminal by using a transmit command word (BC to RT). A screen shot of this command word is shown in Fig. 4.13. Finally, after reception of valid command and data words, remote terminal monitors these data on the monitor interface. Screen shot of the monitor interface is shown in Fig. 4.14. As can be seen in this figure, the transmitted and the received data for the remote terminal are same. It means that, loopback test works without any problem. Cycle diagram of this test is shown in Fig. 4.15.

In this implementation, Bus Controller to Remote Terminal message type was realized

as a data transmission process and Remote Terminal to Bus Controller message type was realized as a data reception process. The other defined message types and detailed error management process for the validation test of this protocol will be implemented as a future work.

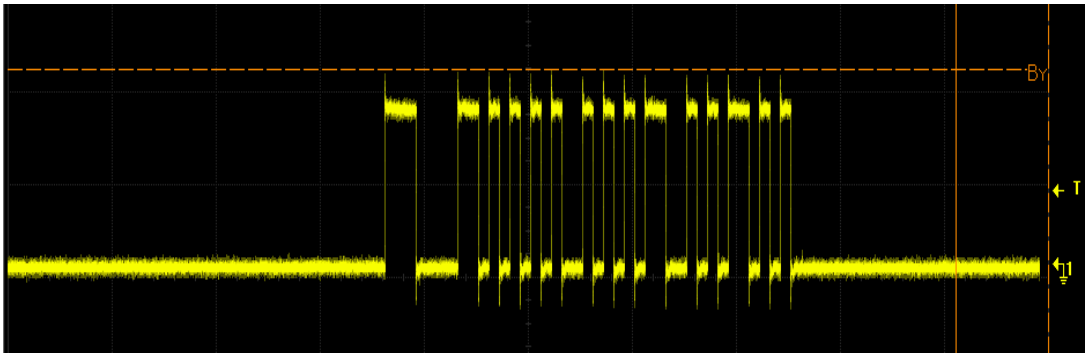

Figure 4.11: Screen shot of the receive command word.

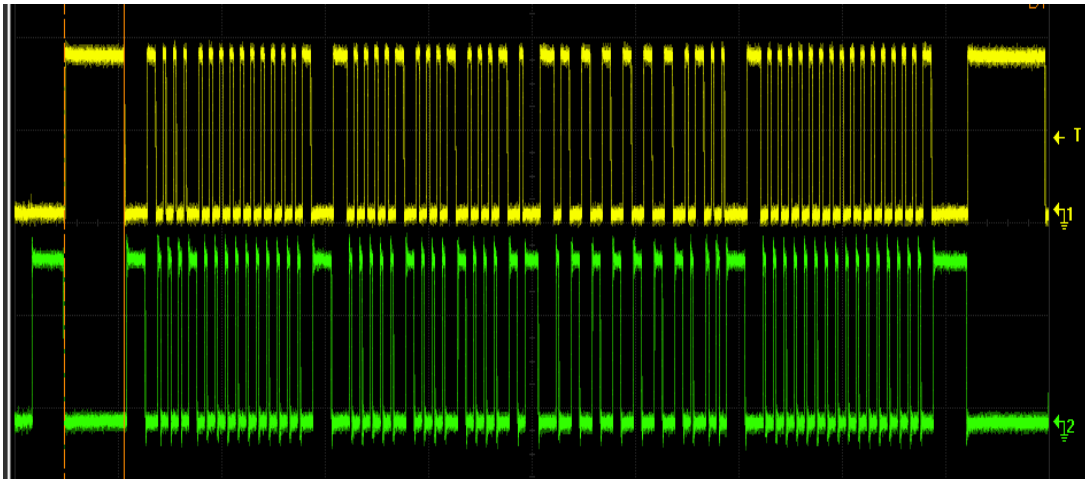

Figure 4.12: Screen shot of the received status and data words.

4.2 Performance Evaluation

The main objective of this work is to design a generic hardware framework for real-time embedded serial bus protocols. The performance of the proposed design was observed through testing five different bus protocols such as MIL-STD-1553B, ARINC-708, ARINC-717, CAN and UART by simulating in MODELSIM platform, compar-

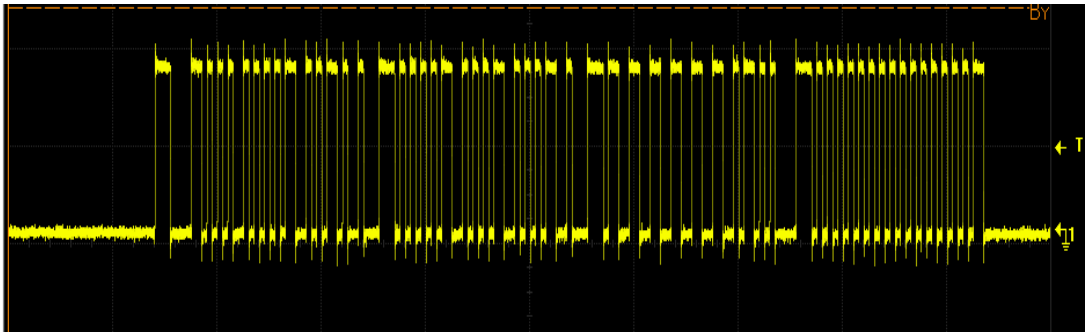

Figure 4.13: Screen shot of the transmit command word.

Control | Mini File System | Kipa | KKS | RT Message Flow | KTS

Flow | Filter

Cnt	Name	Time(s)	T/R	SAd	RtAd	WC	CW	TTag	D[0]	D[1]	D[2]	D[3]	D[4]	D[5]	D[6]	D[7]	D[8]	D
1	Store ID	28,40431609	T	1	15	3	7c23	f65f	0421	5553	0000							
2	AirCraft ID	28,41491264	R	1	15	3	7823	f704	0421	5553	0000							

Figure 4.14: Screen shot of the monitor screen.

ing the performance of the UART implementation with the work proposed in [53] and implementing the bus controller of the MIL-STD-1553B bus protocol as it is mentioned in Design and Implementation section of this chapter.

The efficiency of implemented bus protocols is estimated by the set of criteria: (i) hardware resources required to implement our system; (ii) power consumptions; (iii) quantity of the common blocks in the UNIBUS architecture.

The UNIBUS architecture for the simulation of five different bus protocols was implemented on ISE 14.5 [23] software, and the total on-chip power consumption was estimated by Xilinx's Power Estimator tool [11]. This total power includes both the device dependent static power which is dissipated when there is no signal activity and

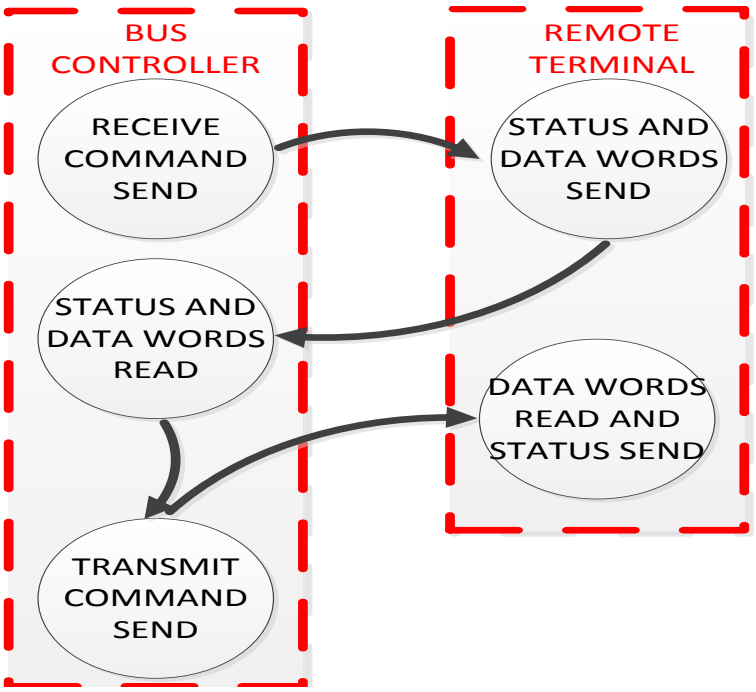

Figure 4.15: Cycle diagram for the loopback test.

the additional dynamic power that is dissipated during the operation.

The hardware resource and power consumption results of these simulations are presented in Table 4.9 and Table 4.10.

Table 4.9: The required hardware resources for the implementation of five different bus protocols.

Bus Protocol	**Number of Slice Flip Flops**	**Number of occupied Slices**	**Total number of 4 input LUTs**	**Number of bounded IOBs**	**Average Fanout of Non-clock Nets**
MIL-STD-1553B	6118	7323	11934	145	3.45
CAN	826	976	1676	35	3.47
ARINC-717	456	623	1015	10	3.44
ARINC-708	10859	9463	10685	76	3.87
UART	357	357	537	18	3.09

When we compare the hardware resources, we see that the implementation of MIL-STD-1553B bus protocol has the highest logic resources (4 input LUTs). This is due to the complexity of this protocol since ten types of frame transmission are realized for it. Beside this protocol, ARINC-708 bus protocol requires the highest number of

Table 4.10: The power consumptions for the implementation of five different bus protocols.

Bus Protocol	Static Power (mW)	Dynamic Power (mW)	Total Power (mW)
MIL-STD-1553B	264	72	336
CAN	263	32	295
ARINC-717	263	28	291
ARINC-708	264	83	347
UART	263	27	290

slice flip flops. This is mostly due to the large embedded registers used to store the required signals since the frame length of this protocol is the highest one which can be seen in in Table 4.1. On the other hand, UART bus protocol consumes the least logic resources (4 input LUTs) while it has the lowest frame length. Therefore, the results show that, the number of the required logic resources is directly proportional to the complexity of the bus protocol.

It can be seen in Table 4.10 that the power consumption of proposed UNIBUS architecture is independent of the type of the implemented bus protocol since the power consumptions of all bus protocols are similar. Therefore, power consumption of the UNIBUS architecture was demonstrated during our implementations. Maximum static power is about 0,265 W, maximum dynamic power is about 0,085 W and maximum total power is about 0,350 W.

Apart from these results, quantity of the common blocks between the implemented five bus protocols attached in Table 4.11 shows the re-usability of the proposed UNIBUS architecture. UNIBUS is designed as generic architecture. Therefore, similarity of the bus protocols is not the important criteria for it. Table 4.11 predicts this conception. It shows that some implemented protocols are similar whereas some other ones are different which is also seen in Table 4.1. However, this condition does not affect the design of the UNIBUS while it has generic architecture. It only affects the logic design of the sub-blocks of the UNIBUS. The logic design of the common blocks seen in Table 4.11 should be same whereas the other sub-blocks should be customized according to the implemented bus protocol.

Table 4.11 shows that the Data Buffer Memory block is only common block for

UNIBUS architecture since it is used as a data buffer between data link layer and physical layer. Commonality of the other blocks in UNIBUS depends on the type of the implemented bus protocol. MIL-STD-1553B and ARINC-708 are the most similar ones since same coding and checksum styles are used as it is mentioned in Fig. **??**. Thus, they have four common blocks. On the other hand, MIL-STD-1553B and CAN are the most different ones since there is only one common block between them.

Table 4.11: The common blocks for the implementation of five different bus protocols.

Common Block	**MIL-STD-1553B**	**CAN**	**ARINC-717**	**ARINC-708**	**UART**
MIL-STD-1553B	-	Data Buffer Memory	Data Buffer Memory, Checksum Generator	Data Buffer Memory, Checksum Generator, Line Decoder, Line Encoder	Data Buffer Memory, Checksum Generator
CAN	Data Buffer Memory	-	Data Buffer Memory	Data Buffer Memory	Data Buffer Memory, Line Decoder, Line Encoder
ARINC-717	Data Buffer Memory, Checksum Generator	Data Buffer Memory	-	Data Buffer Memory, Checksum Generator	Data Buffer Memory, Checksum Generator
ARINC-708	Data Buffer Memory, Checksum Generator, Line Decoder, Line Encoder	Data Buffer Memory	Data Buffer Memory, Checksum Generator	-	Data Buffer Memory, Checksum Generator
UART	Data Buffer Memory, Checksum Generator	Data Buffer Memory, Line Decoder, Line Encoder	Data Buffer Memory, Checksum Generator	Data Buffer Memory, Checksum Generator	-

The performance of UNIBUS architecture was also tested by comparing our UART implementation with the UART implementation in [53]. In this work, a software-hardware method of serial interface controller implementation was proposed. The data link layer functions were implemented in software while physical level functions were implemented in hardware. In order that to estimate the efficiency of offered

method, a system with 1-Wire interface controller based on UART was implemented. Software and hardware modules were implemented directly in FPGA. Spartan-3E FPGA family [25] was used as a hardware device whereas soft microcontroller core PicoBlaze [20] provided by Xilinx company for free was used as a software module.

Two criteria made the efficiency estimation of their UART implementation: (i) the hardware expenses of system implementation; (ii) time interval required for executing of data exchange. They used a histogram to show their results. Each column in their histogram presents the hardware expenses, the size of application software, the size of interface support software and the amount of used general purpose registers.

Same method is used in order to compare our UART implementation results with this work. Comparison results are shown in Fig. 4.16. The x axis of this graph reflects time spent to complete the data exchange process.

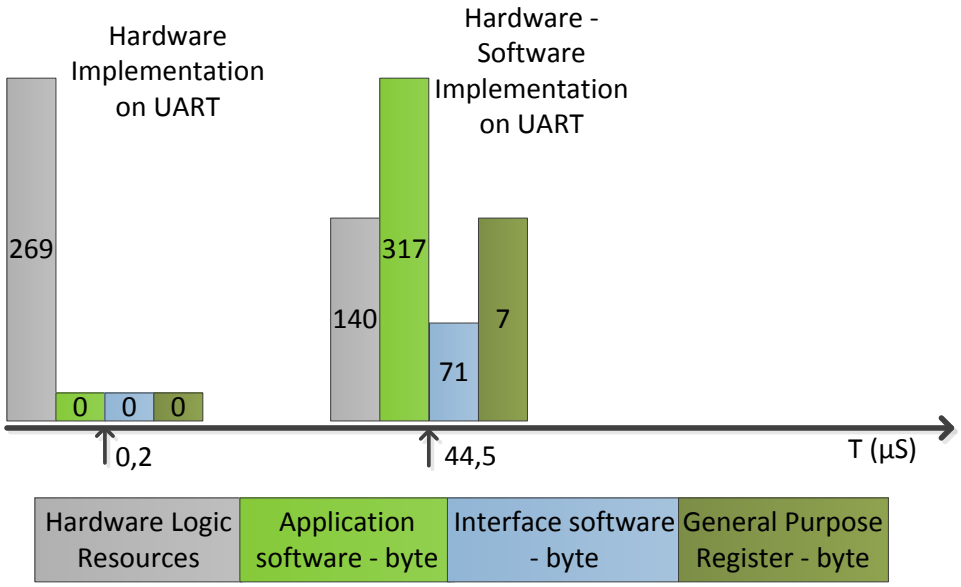

Figure 4.16: The comparison of the UART implementation.

Fig. 4.16 shows that the time of data exchange process for our implementation is about 200 ns whereas the time of data exchange process for the other work is about 44.5 us. Our implementation consists of only hardware architecture, there is no need to application software, interface support software, and general purpose registers.

Because of that reason, our work provides decreasing of the time of data exchange process in order of hundreds. On the other hand, the required hardware logic resources for our implementation is 270 whereas the required hardware logic resources for the other work is about is 140. It means that, our work provides increasing of the required hardware logic resources in 2 times. Xilinx SPARTAN3-XC3S4000-4FG900 [25] FPGA device is used for our implementation. Total amount of logic resources of it is 55296. Thus, the device utilization for our UART implementation is below 1%. Therefore, the amount of hardware expenses of our implementation are not considerable while FPGA supports tens of thousands logic resources.

Finally, we tested the performance of our UNIBUS architecture by implementing it on hardware platform. The bus controller terminal of the MIL-STD-1553B bus protocol was implemented. It was implemented on Xilinx SPARTAN3-XC3S4000-4FG900 [25] FPGA device by using ISE 14.5 [23] software. A loopback test was realized in order to test our design.

We compare our implementation results with the commercial products which are also for MIL-STD-1553B bus protocol. Comparison table is attached in Table 4.12. Five different commercial products such as BRM1553FE, BRM1553D, BRM1553PCI [17],[18], Core1553BBC [21] and Core1553BRM [28] are used for the comparison. Four different criteria such as available modes, area utilization, required memory and required clock frequency are used for this comparison.

First column of this table shows the available modes for the implementation. Our implementation and Core1553BBC product consist of only bus controller terminal of the MIL-STD-1553B whereas BRM1553 and Core1553BRM products support all three terminals of the MIL-STD-1553B. Supporting of all terminals is considered as future work for our design.

Second column of the comparison table shows the area requirements. Our proposed design needs about 3000 logic resources which depend on the FPGA type used in the implementation. This amount is similar to the area requirements of other products. Optimization of the logic design and decreasing the amount of logic resources are considered as future work.

Table 4.12: The comparison between our proposed work and the commercial products.

Products	**Available Modes**	**Area Utilization**	**Required Memory**	**Required Clock Frequency**
BRM1553FE	Bus Controller, Remote Terminal, Bus Monitor	764 to 1059 4-LUT, Depending on FPGA type	32 x 16 bits	Any even frequency of 12Mhz or more
BRM1553D	Bus Controller, Remote Terminal, Bus Monitor	RT only:2800 to 3400 4-LUT Depending on FPGA type; BC + RT + MT:3950 to 4320 4-LUT Depending on FPGA type	2, 4, 8, 16, 32 or 64K x 16 bits	Any even frequency of 12Mhz or more
BRM1553PCI	Bus Controller, Remote Terminal, Bus Monitor	RT only:3100 to 3700 4-LUT Depending on FPGA type; BC + RT + MT:4250 to 4620 4-LUT Depending on FPGA type	2, 4, 8, 16, 32 or 64K x 16 bits; Dual Port RAM	33 or 66MHz
Core1553BBC	Bus Controller	1656 to 2810 cells or tiles Depending on FPGA type; works only with Actel FPGAs	Supports up to 128kbytes of Memory	12, 16, 20, or 24 MHz
Core1553BRM	Bus Controller, Remote Terminal, Bus Monitor	1900 to 8000 cells or tiles Depending on FPGA type; works only with Actel FPGAs	Between 1 kbyte and 128 kbytes (16 bits wide) of internal FPGA memory or external memory used for data storage	12, 16, 20, or 24 MHz
Our Proposed Work	Bus Controller	2000 to 3000 4-LUT, Depending on FPGA type	Limited by FPGA resources (32 x 16 bits	Any frequency larger than 2 MHz (100 MHz)

Third column of the comparison table shows the memory requirements. 32x16 bits memory is used for our implementation. However, our proposed UNIBUS architecture supports larger memory sizes since generic Data Buffer Memory is used in this architecture. The FPGA resources limits the size of this memory. This means larger the FPGA resources, larger the memory size.

Last column of the comparison table shows the clock frequency requirements. We

used different kind of decoding method for our design. By means of this decoding method, our design is tolerant of actual zero-cross distortions and zero crossing errors in the received signal. We capture the zero crossing points in the encoded data instead of using a specific clock signal. However, other products use specific clock frequency, which is equal to the 12 times of the desired data rate, in order to sample and decode the incoming data. Therefore, our proposed design does not need any specific clock frequency while the other products need. Any frequency larger than 2 MHz is enough for our design, since the baud rate of the MIL-STD-1553B protocol is 1 MHz. We used 100 MHz clock signal as the main clock in our implementation.

CHAPTER 5

CONCLUSIONS AND FUTURE WORK

This thesis presents UNIBUS which is a generic hardware architecture to implement the physical and data link layer functions of serial bus protocols. The hardware blocks and the signal interfaces of UNIBUS are designed carefully to separate the protocol specific and protocol independent components to increase the reusability. Such separation also increases the reliability of the design as the protocol independent components which are common among different protocols do not need verification for each implementation. The clear specifications of the protocol specific interfaces also contribute to the reliability. The implementation with UNIBUS is streamlined which decreases the hardware design time.

The versatility of UNIBUS is demonstrated by implementing CAN, UART, ARINC-708, ARINC-717 and MIL-STD-1553B on FPGA using MODELSIM. Furthermore MIL-STD-1553B is fully implemented and tested with a commercial chip. This implementation consists of only bus controller terminal of the MIL-STD-1553B, and it needs about 3000 logic resources which depend on the FPGA type used in the implementation. The resource consumption results show that the generic design of UNIBUS does not decrease the efficiency of the implementations. The results show that, the number of the required logic resources is directly proportional to the complexity of the bus protocol whereas the power consumption of proposed UNIBUS architecture is independent from the type of the implemented bus protocol. Apart from these results, quantity of the common blocks between the implemented five bus protocols were examined. The results show that the dissimilarity of the bus protocols is not an important criterion for our proposed hardware framework. This dissimilarity

condition does not affect the design of the UNIBUS while it has generic architecture. It only affects the logic design of the sub-blocks of the UNIBUS.

The performance of UNIBUS architecture is also tested by comparing UNIBUS UART implementation with the UART implementation in [53]. The results show that, the time of data exchange process is about 200 times smaller in UNIBUS UART by increasing the required hardware logic resources 2 times.

The decoding of UNIBUS is tolerant of actual zero-cross distortions and zero crossing errors in the received signal. The zero crossing points in the encoded data are captured instead of using a specific clock signal. However, other products use specific clock frequency in order to sample and decode the incoming data. Therefore, our proposed design does not need any kind of specific clock frequency different than the other products.

As future work, the design of the remote terminal and bus monitor terminal of the MIL-STD-1553B bus protocol will be added to the proposed design to complete the realization to the commercial product level. In addition, optimization of the logic design and decreasing the amount of logic resources are also considered as a future work. UNIBUS can also be used for the development of new serial bus protocols.

REFERENCES

[1] Osi model, 1999-2001.

[2] Arinc protocol tutorial, June 2000.

[3] Condor mil-std-1553b protocol tutorial, June 2000.

[4] Tlp-1000 series mil-std-1553 transformers, May 2001.

[5] Can bus texas instruments, August 2002.

[6] Ddc mil-std-1553b designer's guide, August 2003.

[7] Hardware design considerations for mil-std-1553 terminals, August 2008.

[8] Hx1188nl pulse transformer, May 2008.

[9] Ti - phyter industrial temperature single port 10/100 mb/s ethernet physical layer transceiver, May 2008.

[10] Understanding and using the controller area network, October 2008.

[11] Xilinx power estimator tool, April 2009.

[12] Aim mil-std-1553b protocol tutorial, November 2010.

[13] Arinc protocol summary, April 2010.

[14] Uart cytron tutorial, December 2010.

[15] Uart exar tutorial, April 2010.

[16] Xilinx platform flash in-system programmable configuration proms, May 2010.

[17] 1553 core types selection, November 2011.

[18] Datasheet of all brm1553 ip cores, November 2011.

[19] Sae mil-std-1553b protocol bus controller validation test, November 2011.

[20] User guide of picoblaze 8-bit embedded microcontroller, June 2011.

[21] Datasheet of core1553bbc, May 2012.

[22] Adesto technologies at25df161-sh-b flash memory, May 2013.

[23] Ise design suite, April 2013.

[24] Pm-db2745s mil-std-1553 transformers, July 2013.

[25] Xilinx - spartan3 series fpga family, June 2013.

[26] Can bus wikipedia, November 2014.

[27] Can protocol tutorial, January 2014.

[28] The handbook of core1553brm, January 2014.

[29] Holt ic - hi-1579 mil-std-1553b bus protocol tranceiver chip, May 2014.

[30] Holt ic - hi-6121 mil-std-1553b bus protocol remote terminal chip, July 2014.

[31] Modelsim - advanced simulation and debugging platform, July 2014.

[32] Ti - tms320c6747 fixed/floating-point digital signal processor, June 2014.

[33] Uart wikipedia, November 2014.

[34] W. Ahmad, Z. Shengbing, H. Amjad, G. Gillani, and A. Jianfeng. Fpga based real time implementation scheme for arinc 659 backplane data bus. In *Computer Research and Development (ICCRD), 2011 3rd International Conference on*, volume 4, pages 478–482, March 2011.

[35] G. W. An Yu Cheng, Ju Lin Zhang. Design of vehicle can-network bit-level interference testing system based on fpga. In *Advanced Materials Research, 2013*, volume 765 - 767, pages 129–133, September 2013.

[36] G. W. An Yu Cheng, Ju Lin Zhang. Performance analysis of controller area network. In *Applied Mechanics and Materials , 2014*, volume 543-547, pages 3499–3502, Marc 2014.

[37] J. Artal, J. Caraballo, and R. Dufo. Can/lin-bus protocol. implementation of a low-cost serial communication network. In *Tecnologias Aplicadas a la Ensenanza de la Electronica (Technologies Applied to Electronics Teaching) (TAEE), 2014 XI*, pages 1–8, June 2014.

[38] T. Blessington, B. Murthy, G. Ganesh, and T. Prasad. Optimal implementation of uart-spi interface in soc. In *Devices, Circuits and Systems (ICDCS), 2012 International Conference on*, pages 673–677, March 2012.

[39] Q. Cai, Y. Guo, W. Chen, and M. Wang. A programmable controller based on can field bus embedded microprocessor and fpga. volume 7129, pages 712919–712919–6, 2008.

[40] L. Chen, J. Lei, Q. Wang, N. Lin, X. Long, and C. Hou. A novel method for evaluating the attenuation of 1553b bus system. In *Computer Science Education (ICCSE), 2012 7th International Conference on*, pages 281–284, July 2012.

[41] C. Hou, S. Wang, Q. Wang, and H. Zhang. Performance analysis of high-speed mil-std-1553 bus system using dmt technology. In *Computer Science Education (ICCSE), 2013 8th International Conference on*, pages 533–536, April 2013.

[42] Z. HU, J. ZHANG, and X. ling LUO. A novel design of efficient multi-channel {UART} controller based on {FPGA}. *Chinese Journal of Aeronautics*, 20(1):66 – 74, 2007.

[43] J. H. Hua Zang, Hai Wei Mu. Design and implementation of intelligent controller based on can bus of charge and discharge machine for lead-acid battery. In *Advanced Materials Research, 2014*, volume 926-930, pages 1257–1260, May 2014.

[44] J. Jaeger. FPGA-based prototyping grows up. *Electronic Engineering Times*, (518), 2008.

[45] M. Jayananda and N. Jayarathne. Development of a field programmable gate array based controller area network sniffer. In *Industrial and Information Systems (ICIIS), 2013 8th IEEE International Conference on*, pages 610–615, Dec 2013.

[46] T. Jena, A. Swain, and K. Mahapatra. A novel bit stuffing technique for controller area network (can) protocol. In *Advances in Energy Conversion Technologies (ICAECT), 2014 International Conference on*, pages 113–117, Jan 2014.

[47] Y. Jiang, B. Liang, and X. Ren. Design and implementation of can-bus experimental system. In *Strategic Technology (IFOST), 2011 6th International Forum on*, volume 2, pages 655–659, Aug 2011.

[48] J. Jose. Design of manchester ii bi-phase encoder for mil-std-1553 protocol. In *Automation, Computing, Communication, Control and Compressed Sensing (iMac4s), 2013 International Multi-Conference on*, pages 240–245, March 2013.

[49] J. Jose and S. Varghese. Design of 1553 protocol controller for reliable data transfer in aircrafts. In *Intelligent Systems Design and Applications (ISDA), 2012 12th International Conference on*, pages 686–691, Nov 2012.

[50] N. Jusoh, A. Ibrahim, M. Haron, and F. Sulaiman. An fpga implementation of shift converter block technique on fifo for uart. In *RF and Microwave Conference (RFM), 2011 IEEE International*, pages 320–324, Dec 2011.

[51] B. Kirk. FPGA-prototyping and ASIC-conversion considerations. *EDN*, 52(21):67 –70, 2007.

[52] D. Li-feng, D. Ming, and L. Jian-ming. Application of ip core technology to the 1553b bus data traffic. In *Microelectronics and Electronics (PrimeAsia),*

2010 Asia Pacific Conference on Postgraduate Research in, pages 338–342, Sept 2010.

[53] I. Maykiv. The method of software-hardware implementation of serial interfaces. In *Modern Problems of Radio Engineering, Telecommunications and Computer Science, 2008 Proceedings of International Conference on*, pages 439–441, Feb 2008.

[54] M. Mostafa, M. Shalan, and S. Hammad. Fpga-based low-level can protocol testing. In *System-on-Chip for Real-Time Applications, The 6th International Workshop on*, pages 185–188, Dec 2006.

[55] C. Pantiruc and M. Negru. Fpga based can data visualization. In *Intelligent Computer Communication and Processing (ICCP), 2011 IEEE International Conference on*, pages 245–252, Aug 2011.

[56] T. Presi. Design and development of pic microcontroller based vehicle monitoring system using controller area network (can) protocol. In *Information Communication and Embedded Systems (ICICES), 2013 International Conference on*, pages 1070–1076, Feb 2013.

[57] R. Randhawa and M. Imran. A low cost design of mil-std-1553 devices. In *Satellite Telecommunications (ESTEL), 2012 IEEE First AESS European Conference on*, pages 1–4, Oct 2012.

[58] A. . L. N. Roopesh N.M. Vhdl implementation of 1553 protocol using actel ip core. In *International Journal of Advanced Computer Engineering and Communication Technology (IJACECT)*, pages 2278–5140, Feb 2013.

[59] C. C. Shan Shan Zhang. Design and implementation of intelligent multi-serial ports data transmission system based on nios ii using in smart home. In *Applied Mechanics and Materials, 2014*, volume 556 - 562, pages 1605–1609, May 2014.

[60] S. SURESH. Vhdl implementation of manchester encoder and decoder. In *International Journal of Electrical, Electronics and Data Communication*, pages 2084–2320, Apr 2013.

[61] K. Szurman, J. Kastil, M. Straka, and Z. Kotasek. Fault tolerant can bus control system implemented into fpga. In *Design and Diagnostics of Electronic Circuits Systems (DDECS), 2013 IEEE 16th International Symposium on*, pages 289–292, April 2013.

[62] G. Wakhle, I. Aggarwal, and S. Gaba. Synthesis and implementation of uart using vhdl codes. In *Computer, Consumer and Control (IS3C), 2012 International Symposium on*, pages 1–3, June 2012.

[63] H. Xiang-Dong, Y. Hui-mei, and Z. Xiao-Xu. Design of dual redundancy can-bus controller based on fpga. In *Industrial Electronics and Applications (ICIEA), 2013 8th IEEE Conference on*, pages 843–847, June 2013.

[64] Z. B. S. Xiao Rong Tong. Design of uart with crc check based on fpga. In *Advanced Materials Research, 2012*, volume 490 - 495, pages 1241–1245, March 2012.

[65] J. Yousaf, M. Irshad, and I. Mehmood. Implementation of 1553b bus protocol on fpga board using digital phase lock loop. In *Emerging Technologies (ICET), 2012 International Conference on*, pages 1–6, Oct 2012.

[66] L. Zhijian. Notice of retraction research and design of 1553b protocol bus control unit. In *Educational and Network Technology (ICENT), 2010 International Conference on*, pages 330–333, June 2010.

[67] Y. Zhu, Y. Wang, and U. Schaefer. Study on the communication between fpga and observer using controller area network and uart. In *Information Networking and Automation (ICINA), 2010 International Conference on*, volume 1, pages V1–240–V1–244, Oct 2010.