POLITECNICO DI TORINO

Collegio di Ingegneria Elettronica, delle Telecomunicazioni e Fisica (ETF)

Corso di Laurea Magistrale in
Ingegneria Elettronica (Electronic Engineering)

Tesi di Laurea Magistrale

Custom High-Speed Communication Ethernet-Point to Point Protocol Interface Design and Implementation

Relatore
Prof. Guido Masera

Supervisori
Jorge Sanchez de Nova
Dr. Javier Garçia Castaño

Candidato

Elena Maria Migliorin

Dicembre 2018

Abstract

This master thesis work describes the development of a high-speed protocol interface for the railway communication system at Bombardier Transportation Sweden AB. The railway communication system helps the maintenance, controls, and monitors all the objects involved in a railway line such as traffic lights and sensors. The communication network is normally composed by a physical part, such as wires and processors, and the firmware and software to manage the traffic of data. The data transmitted are the commands that the objects on the railway line must execute and their operational status.

High-speed communication in a railway network increases the control of the objects on the railway line in terms of response time. A fast dispatching of orders increases the reactivity of the system to any event affecting the railway objects. This improved reactivity allows for an increase of train ride per hour, improving the transportation service for the customers.

This master thesis work was carried out at Bombardier Transportation, Stockholm, which is a world leader for aerospace and transportation. The railway communication network at Bombardier Transportation monitors and drives all the elements on the railway line. The cross point of two railways must be synchronized to make the fastest transit of the trains. If the communication toward the traffic light is fast enough, the trains will avoid waiting for a long time before transiting the cross point. The communication network features physical channel that can support high-speed communication. In the constant effort to improve the service for the customer by reducing the waiting time between train rides, Bombardier is trying to upgrade the inter-board communication. The plan is to use a new physical channel which is estimated to be 100 times faster than the currently used one. This new channel needs a suitable firmware interface to allow its use in the existing system.

Care must be taken when designing this interface to ensure high reliability and safety. The System Integrity Level (SIL) is the standard that defines four discrete level of safety integrity requirements. In the railway system, the SIL is 4, the highest. According to the standard, only one serious error is allowed in 10000 years. Since the safety regulation is so tight, it means that the corruption of a single message in railway communication can result in serious safety hazards. Therefore, all the system's elements must comply with the RAMS characteristic: Reliability, Availability, Maintainability, and Safety.

In this thesis, I developed a custom-made Very-High-Speed Integrated Circuits Hardware Description Language (VHDL) interface to use the new physical channel for high-speed communication while maintaining standard safety levels. This interface converts the message between the standard protocol in the system (Ethernet) and the communication protocol chosen for the new channel, which is the Point to Point Protocol (PPP). This protocol was chosen since it has a flexible header and provides Cyclic Redundant Check (CRC) code, which makes the communication safe and adaptable. The interface must also be divided into two main parts since the communication channel is full duplex. When a message is received by the board it will enter the inbound part. If it is sent from the board it will be converted by the outbound part. The inbound part then converts the frame from PPP to Ethernet and the outbound from Ethernet to PPP.

The design flow of the component is composed of four main stages. The existing system was studied in terms of component, signals and timing issues since the new component must be synchronized with the whole chain of transmission. Then the component was developed with the goal of creating a fast component and keeping the integrity of the sent frames. The generated VHDL code was then stimulated with custom testbenches to verify the behavior of all the developed parts. Finally, the code was uploaded on the physical board and the system communication was tested.

As first step, all the components belonging to the transmission chain have been studied. All of the described in VHDL and some are network standard. They have been analyzed in terms of timing synchronization, the dimension of the data and how they are handled by these components and what is their role in the system. This study is essential to understand the requirement that the system has in terms of synchronization toward the new component.

The PPPConverter was then developed analyzing how the two parts must be interfaced toward the system and how the conversion has to be managed from one kind of frame to the other. As first step, a pseudocode was written to show the behavior of each part. Then the flow of the data was studied. Since one important parameter is the performances of the system, the writing and reading operation are done as parallel as possible. Moreover, since the integrity of the signal is essential, the PPP frame header is controlled in each byte. If a faulty byte is read, the whole frame is discarded. Moreover, a shift register, called ShiftMemory, was implemented to determine the end of the payload of both frames. In both parts, the length of the payload was not available and with this component, it is possible to eliminate the incoming end-of-frame bytes and write the new ones. Timing diagrams were also studied and developed to satisfy the interface of the component toward the system and toward the new channel, called IOChannel.

To verify the correct behavior of the generated VHDL code, the ShiftMemory and the two parts were stimulated. The test was developed with custom-made testbenches and the generated waveforms were confronted with the expected timing diagrams. These test also aimed to verify the full-duplex ability of the PPPConverter, since the inbound and outbound parts should work in parallel independently from each other.

Finally, the new VHDL code was uploaded on the physical board to verify its behavior. At first, the full transmission chain was simulated. Using a testbench, the simulation recreated the communication of a random ARP message from one board to another via the IOChannel. Then, only the IOChannel specific interface was uploaded on the board to verify that the physical channel was actually working. Then the interface was uploaded together with the old system but in parallel. This step was implemented to verify that the extra logic could still be handled by the existing system. Finally, the whole desired implementation was uploaded on the board to verify that the developed component could create a new communication path between the system and the IOChannel.

The entire PPP protocol is not fully implemented: future implementations will need to add the Escaping mechanism as declared by the standard. Moreover, the destination and source addresses belonging to the Ethernet frame need to be managed in a more specific way.

Sommario

Questo elaborato di tesi descrive lo sviluppo di un'interfaccia tra protocolli di comunicazione ad alta velocità per il sistema di comunicazione ferroviario sviluppato in Bombardier Transportation. Il sistema di comunicazione in ambito ferroviario controlla, monitora e aiuta la manutenzione di tutti gli oggetti presenti su una linea ferroviaria come semafori e sensori. La rete di comunicazione è solitamente composta di una parte fisica, come cavi e processori, e dal firmware e software utilizzati per dirigere il traffico di dati. I dati trasmessi sono i comandi che gli oggetti sulla linea ferroviaria devono eseguire e il loro stato operativo.

La comunicazione ad alta velocità in una rete ferroviaria migliora il controllo degli oggetti sulla linea ferroviaria in termini di tempi di risposta. Un invio di ordini più veloce aumenta la reattività del sistema a qualsiasi evento del quale gli oggetti ferroviari sono affetti. L'aumentata reattività permette un aumento di corse di treni all'ora, migliorando il servizio di trasporto per i clienti.

Questa esperienza di tesi è stata sviluppata presso la Bombardier Transportation, a Stoccolma, che è un leader mondiale per il settore aerospaziale e i trasporti. La comunicazione ferroviaria sviluppata dalla Bombardier Transportation monitora e pilota tutti gli elementi sulla linea ferroviaria. Il punto d'incrocio di due linee ferroviarie deve essere sincronizzato per permettere il passaggio dei treni il più velocemente possibile. Se la comunicazione verso il semaforo è sufficientemente veloce, il treno eviterà di aspettare per lungo tempo prima di transitare nell'incrocio. La rete di comunicazione presenta canali fisici che supportano la comunicazione ad alta velocità. Nel costante sforzo di migliorare il servizio per i clienti nel ridurre i tempi di attesa fra corse dei treni, Bombardier Transportation sta provando a migliorare la comunicazione fra board interne al sistema. L'obiettivo è di usare un nuovo canale finisco che si è stimato essere 100 volte più veloce di quello correntemente in uso. Il nuovo canale necessita un'interfaccia adatta per permettere di usarlo nel sistema esistente.

Quando si progetta questa interfaccia, è necessario farlo con cura per garantire un'alta affidabilità e sicurezza. Il System Integrity Level (SIL) è lo standard che definisce Quattro livelli discrete di requisiti sull'integrità di sicurezza. Nel sistema ferroviario il SIL è 4, il massimo valore. Secondo lo standard, solo un errore significativo è ammesso in 10000 anni. Poiché la regolazione di sicurezza è severa, significa che la corruzione di un solo messaggio nella comunicazione ferroviaria può generare serie situazioni di pericolo. Di conseguenza, tutti gli elementi del sistema devono rispettare le caratteristiche RAMS: Reliability (Affidabilità), Availability (Disponibilità), Maintainability (Facilità nella manutenzione) and Safety (Sicurezza).

In questa tesi, ho sviluppato un'apposita interfaccia in Very High-Speed Integrated Circuits Hardware Description Language (VHDL) per usare il nuovo canale per la comunicazione ad alta velocità mantenendo I livelli di sicurezza standard. Questa interfaccia converte il messaggio tra il protocollo standard del sistema (Ethernet) e il protocollo di comunicazione scelto per il nuovo canale, il Point to Point Protocol (PPP). Questo protocollo è stato scelto perché ha un header flessibile e ha il Cyclic Redundant Check (CRC), che rendono la comunicazione sicura e adattabile. L’interfaccia deve anche essere divisa in due parti principali poiché il canale di comunicazione è full duplex. Quando un messaggio è ricevuto dalla board, entrerà nella parte inbound. Se è inviato dalla board, sarà convertito dalla parte outbound. La parte inbound converte quindi il frame da PPP a Ethernet e l’outbound da Ethernet a PPP.

Il flusso di progetto del component è composto di quattro fasi. Il sistema esistente è stato studiato in termini di component, segnali e caratteristiche temporali perché il nuovo component deve essere sincronizzato con tutta la catena di trasmissione. Il component poi è stato sviluppato con l’obiettivo di creare un component veloce mantenendo l’integrità del frame inviata. Il codice VHDL generato è stato poi simulato con testbench appositi per verificare il comportamento di tutte le parti sviluppate. Infine, il codice è stato caricato sulla board fisica e il sistema di comunicazione è stato testato.

Come primo passo, tutti i component appartenenti alla catena di trasmissione sono stati studiati. Essi sono descritti in VHDL e molti sono standard di rete. Sono stati analizzati in termini di sincronizzazione temporale, dimensione dei dati e come sono gestiti. Anche il loro ruolo nel sistema è stato analizzato. Lo studio è essenziale per capire I requisiti che il sistema ha in termini di sincronizzazione verso il nuovo component.

Il PPPConverter è stato sviluppato analizzando come le due parti debbano essere interfacciate verso il sistema e come la conversione da un tipo di frame all’altro debba essere gestita. Come primo passo, uno pseudocodice è stato scritto per mostrare il comportamento di ogni parte. Poi il flusso di dati è stato studiato. Dato che un parametro importante sono le performance del sistema, le operazioni di scrittura e lettura sono svolte il più possibile in parallelo. Inoltre, dato che l’integrità del segnale è essenziale, l’header del frame PPP è controllato in ogni suo byte. Se un byte sbagliato è letto, l’intero frame è scartato. In aggiunta, uno shift register, chiamato ShiftMemory, è stato implementato per determinare la fine del payload di entrambe i frame. In ambo le parti, la lunghezza del payload non è disponibile e con questo component è possibile eliminare i byte di fine frame e scrivere i nuovi. Sono anche stati studiati I diagrammi temporali per soddisfare l’interfaccia del component verso il sistema e il nuovo canale, chiamato IOChannel.

Per verificare la correttezza del comportamento del codice VHDL generato, lo ShiftMemory e le due parti sono state simulate. Il test è stato sviluppato con appositi testbench e le forme d’onda generate sono state confrontate con i diagrammi temporali attesi. Questi test avevano come obiettivo anche quello di verificare la capacità di essere full duplex del PPPConverter, dato che le parti inbound e outbound devono lavorare in parallelo.

Infine, il nuovo codice VHDL è stato caricato nella board fisica per verificarne il comportamento. Come prima cosa, l’intera trasmissione è stata simulata. Tramite un testbench, la simulazione ha ricreato la comunicazione di un messaggio ARP casual da una board all’altra attraverso l'IOChannel. Poi, solo l’interfaccia specifica per l’IOChannel è stata caricata sulla board per verificare che il canale fisico stesse effettivamente lavorando. L’interfaccia poi è stata caricata insieme al vecchio sistema ma in parallelo. Questo passaggio è stato svolto per verificare che la logica aggiuntiva potesse essere gestita dal sistema esistente. Infine, l’intera implementazione richiesta è stata caricata sulla board per verificare che i component sviluppati potessero creare un nuovo passaggio di comunicazione tra il sistema e l’IOChannel.

L’intero protocollo PPP non è stato implementato del tutto: sviluppi futuri richiedono di aggiungere il meccanismo di escaping come dichiarato dallo standard. In aggiunta, gli indirizzi di provenienza e destinazione appartenenti al frame Ethernet devono essere gestiti in maniera più specifica.

Indice

1. Introduction

A telecommunication network is a system that allows different elements to communicate with each other at a distance using electromagnetic signals[1]. Communications systems can be classified based on the way the communication work. The communication can either work so that all the objectives can communicate with each other simultaneously, or it can be limited to a message from a single object per communication line at the time. Whenever the latter case occurs, it is possible to set a hierarchy among the objects of the system in a Master-Slave fashion. It is possible to assign the role of Master of the communication to a specific object. A Master, in short, can decide when to send the messages and who the receiver is. An element that is not a Master is a Slave of communication, Slave in short. A Slave can talk only when a Master allows it to do it. In complex systems, there is the need to use different levels of hierarchy. The Slaves can be Masters as well, totally passive elements or can send messages to the Master which is going to listen to them using a specific priority order.

Telecommunication networks are used in railway transportation to coordinate the traffic of the trains. The management of the traffic of the trains in each station requires the precise coordination of many electromechanical devices both on the railway and onboard the train. The traffic must be handled by a central traffic unit called Traffic Management System (TMS) which enforces the routes and the schedule of the trains and takes the decisions when two or more trains must transit on the same railway line. The TMS communicate with the Computer-based Interlocking System (CBI), which is responsible for the supervision and control of the object in the railway line. The CBI is composed of three subsystems: the Central Interlocking System (CIS), the Transmission Network (TN) and the Object Controller System (OCS) (Figure 1.1). When a train has the allowance to transit on a line, the TMS sends commands through the communication network until they reach the traffic lights and question the position sensors status. Traffic light and sensors are some of the objects on the railway line, also called Wayside Objects. Wayside Objects and they are traffic lights, balises, sensors, and points. When the TMS has decided which Wayside Object should move, the order is communicated to the CBI that takes it in charge and it is responsible for the direct control of the Wayside Objects. The CBI is composed of three subsystems: the Central Interlocking System (CIS), the Transmission Network (TN) and the Object Controller System (OCS). The OCS is the element that receives the order from the TMS and translates them into messages that the objects on the railway can understand and execute. The CIS is the communication node between the TMS and the OCS. The TN is the system built of all the elements that compose the network that make the CIS-OCS communication possible.

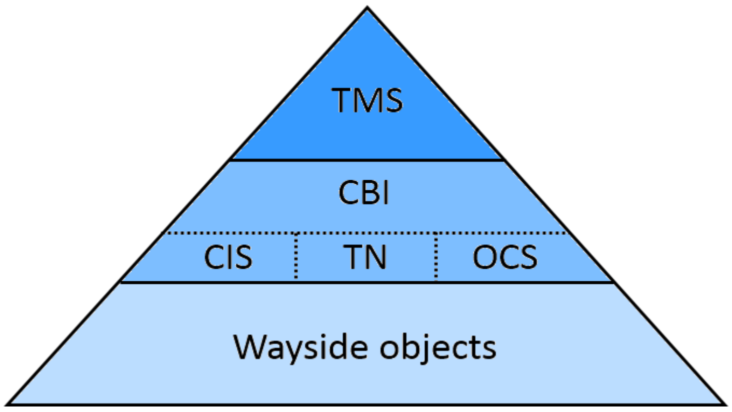


Figure 1.1: The hierarchical structure of the Railway Communication System. The TMS communicates with the CBI. The CBI is composed by the CIS, TN and the OCS which is the elements that directly control the Wayside Objects.

Important factors influencing the wellness of customers of the railway transportation system are the compliance with the highest safety standard and high-speed communication. The safety standard is regulated by the System Integrity Level (SIL). It is defined as a relative level of risk-reduction provided by a safety system. In the railway system, the required SIL is 4, the highest level defined by the standard. The standard requires at Level 4 a probability of failure per hour of $10^{-8} – 10^{-9}$, or one failure every 10000 years, for a system operating in continuous mode. The Wayside Objects have to be constantly monitored about their status to prevent damage and react quickly to possible hazardous situations. The high-speed communication has an important role in improving the quality of transportation. If the communication network is fast, the Wayside Object can react in a short period of time, and the overall system will have better performances. If the order to change the point machine arrives faster, the same line will have more frequent train rides resulting in reducing waiting time for the customer.

A new high-speed channel will be used to increase locally the performance of communication inside the OCS. In the OCS subsystem, there are several elements that concur to the flow of the messages from and toward the Wayside Objects. The message that comes from the CIS is sent to the OCS. The communication network between the CIS and the OCS is also composed of two redundant physical boards. These boards are identical and they are defined as redundant because one of them is designated as *online* and the other one is in *standby* mode. Whether the *online* one stops working, the system substitutes it with the *standby*. In this way, communication is always granted even when a hazard occurs. Currently, communication from the *online* board to other elements of the OCS occurs through a specific channel. However, the performances of this channel are not high enough to update the system to increase the railway traffic. A high-speed physical channel has to be used. This channel is already present on the Printed Circuit Board (PCB) inside the OCS, but it was missing a firmware interface to allow it to communicate with the current system. It has been proven that this new channel is 100 times faster with respect to the previous one[2]. Using the new channel, the traffic of the messages can be increased.

It is important that the design of the interface that will allow the communication through the new channel is custom made to fit the system already developed inside the FPGA. The new channel needs to be connected to the existing system and a proper interface has to be developed to enable a standard communication and to comply speed and safety requirements. The interface is designed using VHDL language and then loaded into a Field Programmable Gate Array (FPGA)[3]. An FPGA is a device that is composed of a reprogrammable logic. Each board is composed by an FPGA which acts as a Switch[4], which is a device that redirects the different incoming messages to the correct destination device. The channel is directly connected to the FPGA on the PCB. The message arrives in the FPGA and it dispatches it to the proper destination. An on-the-shelf solution was not a good choice for this system since the OCS developed by Bombardier Transportation is granted for 30 years. In that timespan, the device can become obsolete and the integrity of the system might not be granted since an obsolete device can grant less reliability with respect to a new one. It is then a flexible solution and it facilitates maintenance operations since it can be reconfigured with new coding instead of replacing a new component on the board.

The new interface enables communication between two different standard protocols and it is custom made to comply with the existing system. The standard protocol that the system uses for the messages is the Ethernet [6](see Chapter 2, Section 2.5). To analyze if the Ethernet frame was still a good choice in terms of protocol to be used on the new channels, the flexibility, the speed, and safety of the system were taken into consideration. A frame is the transmission unit which carries the message by encapsulating it with a header and an end-of-frame field as the start and stop points of the message. The carried message is called the payload. The communication is faster if more payload information is transferred in the same amount of time, and therefore if the frame has less non-payload bytes to send. To keep the communication fast, a frame which has less overhead with respect to the Ethernet has to be considered. In this work, the overhead is defined as the ratio between the length of the non-payload bytes and the total length of the frame and it is the parameter to consider whether the chosen protocol fits the high-performance requirement. A low overhead value results in a more efficient information transfer, decreasing the time-per-message and thus allowing faster communication. To find a valid alternative to the Ethernet frame and since the communication is serial, also the Serial Line Interface Protocol (SLIP) and the Point to Point Protocol (PPP)[7] have been studied (for the complete frame explanation, see Chapter 2 Sections 2.6 and 2.7). The SLIP is a protocol developed to give a start and end frame of a message to be sent over a serial line. The PPP is a protocol is meant to connect two elements that have only one link between them. It is possible to notice that in the worst case scenario, that is when the frame has the maximum dimension, the best solution is the SLIP. However, the PPP offers two fields that are not available in the SLIP[8]: the header bytes that precede the payload and the Cyclic Redundant Check (CRC) [9] field. These fields allow having a flexible communication that is fully customizable according to the need of the receiver of the message and to have an extra security layer by exploiting the CRC code (see Chapter 2, Section 2.3). The new interface then will be a converter between Ethernet and PPP frame and it will be called PPPConverter. It will be composed of two separated and independent part called as inbound and outbound. The part will be called Inbound if the message in incoming in the board, outbound if it sent from the board over the IOChannel, the new channel.

By the physical point of view, the new channel, called IOChannel, is implemented as an LVDS-full duplex communication link. A full duplex communication means that the system can exchange messages from and to the board on two independent links. Moreover, each link is a Low Voltage Differential Signal (LVDS)[5]. This means that each link is composed of a positive and a negative line. Therefore, there are four lines for one channel, two are with the message as it is, defined as *positive,* and two in which the data are sent as the opposite value called *negated*. For further information about the LVDS protocol, see Chapter 2 Section 2.2. Each wire sends messages using a serial protocol implemented by a Universal Asynchronous Receiver-Transmitter (UART) component inside the FPGA (described in Chapter 2, Section 2.4). This asynchronous component allows to serialize and send the data over the lines and deserialize the received bits. The serial UART data generated by the architecture inside the FPGA reaches a buffer that creates a differential signal as output and then the output is transmitted onto the LVDS pins of the FPGA. When the message is received, it is translated from LVDS to single ended from another buffer inside the FPGA and it is converted into a byte-based message from the UART component.

In conclusion, an Ethernet-PPP converter has been designed and implemented in VHDL to allow communication through the IOChannel. The interfaces connecting the IOChannel and the system will be custom made according to the given specifications to interface this component with the existing system and the UART module that acts as a transceiver toward the physical implementation of the IOChannel. The component is working as required from a behavioral point of view. When integrated into the existing system on a physical board, the system does not produce anymore the output on the IOChannel.

2. Bombardier Transportation

The thesis project was developed and implemented at Bombardier Transportation Sweden AB. Founded as *L'Auto-Neige Bombardier Limitée* in 1942 by Joseph-Armand Bombardier, Bombardier is a multinational aerospace and transportation company based in Montreal, Quebec, Canada. In 1970, Bombardier buys Lohnerwerke in Vienna, Austria, a manufacturer of motor scooters and trams, and its subsidiary, the engine manufacturer ROTAX. This marks Bombardier's entry into the railway business. After obtaining the North American leadership in Rail Transportation in 1982, in 1986 Bombardier expands in Europe and in the same year it buys Canadair expanding his sectors also in the aerospace field [10].

Nowadays, Bombardier Transportation is one of the largest companies in the world in the rail vehicle and equipment manufacturing and servicing industry. Among many products, Bombardier has produced many subway systems worldwide, monorails, trams, and rail vehicles. Famous products are the train V300ZEFIRO, also known as Frecciarossa 1000, shown in Figure 2.1 [11] and the Metro trains and system in Munich Airport Franz Josef Strauss, INNOVIA APM 300 Automated People Mover System, shown in Figure 2.2 [12].

Figure 2.1: V300ZEFIRO

Figure 2.2: INNOVIA APM 300

3. The involved standard protocols, models, and utilities

Several standard protocols, model and utilities belonging to telecommunication engineering have been considered during the development of this thesis project. They include Ethernet, PPP, SLIP and ARP protocols, the Ping utility, the OSI Model, the UART device, the LVDS standard, and the CRC code.

3.1 Open System Interconnection (OSI) model

The Open System Interconnection model is a standard for network development[13]. It describes the architecture of a network system as composed of seven different layers whose set fully describes all the possible functionality of a network system. For this project, the FPGA works only in the OSI Layer 2[14] while transporting messages belonging to the Layer 3. It only deals with the correct addressing and transmission of the messages but not their content.

3.1.1 Layer 1: Physical Layer

The Protocol Data Unit (PDU), which is a single unit of information transmitted among peer entities of a computer network, is organized in *symbols*. This layer defines the way the raw bits are transmitted. It provides the electrical, mechanical and procedural interface to the transmission medium. It is often referred to as PHY.

3.1.2 Layer 2: Data Link Layer

The PDU is the *frame* for this layer. It focuses on delivering, addressing and arbitering the media of frames usually in a Local Addressing Network (LAN). It is not managed the local network, it is handled by higher levels. It is divided into two sub layers[14]: logical link control (LLC) and media access control (MAC). The former multiplexes the protocols that run in the data link layer and can provide flow control, acknowledgment and error notification as the checksum control. The latter handles frame synchronization and the arbitration for speaking or listening on a line. Some protocols belonging to this layer can be Ethernet[6], PPP[7], and SLIP[8].

3.1.3 Layer 3: Network Layer

The *packet* is the PDU of this layer. The Layer 3 transfers variable length packets from a source to a destination host via one or more networks. It takes order from the Transport Layer and gives orders to the Data Link Layer.

3.1.4 Layer 4: Transport Layer

The protocols that work in this layer provide host-to-host communication services. The PDU for this layer is *segment* or *datagram*.

3.1.5 Layer 5: Session Layer

The Session Layer provides the mechanism for opening, closing and managing a session between application processes. The PDU for this layer is *data*.

3.1.6 Layer 6: Presentation Layer

The presentation layer is responsible for the delivery and formatting of information to the application layer for further processing or display.

3.1.7 Layer 7: Application Layer

The application layer is an abstraction layer that specifies the shared communications protocols and interface methods used by hosts in a communications network.

3.2 Low-Voltage Differential Signaling (LVDS)

LVDS is a standard that specifies the electrical characteristics of a differential and serial communication protocol. It is a protocol defined in the OSI Layer 1. Each signal in this standard is sent via two wires which are called *positive* and *negative*. Due to this differential characteristic, the common mode is rejected.[5]

3.3 Cyclic Redundant Check (CRC)

The CRC is an error detecting code used in order to observe an accidental change in the data inside the frame [15]. It is usually used in digital network systems by many protocols as part of their frame as Ethernet, PPP, and RapidIO. The CRC can also be used for error correction but in this project will be used only the CRC32 [9] error detection version. CRC32 is the algorithm that has 32 bits as input and 32 as output. The CRC uses a specific polynomial to which the input frame is divided using a polynomial division and creating a four bytes sequence unique for the frame to which it is related. All the parts in the frame as header, payload and possibly the tail will be set as the input of the CRC32.

3.4 Universal Asynchronous Receiver-Transmitter (UART)

A UART is a device used for serial communication over a serial port. This module divides the frame into bits to be sent one per each clock cycle on the serial line. The delimiters of the sent byte are the start and stop bit. The start bit is determined by a transition from 1 to 0, then eight bits are transmitted, and the stop bit is 1 as shown in Figure 3.1. The communication is asynchronous: the UART receiver should work at the same frequency of the transmitter or use an asynchronous buffer to not lose data. In this project, the UART modules work at the same frequency.

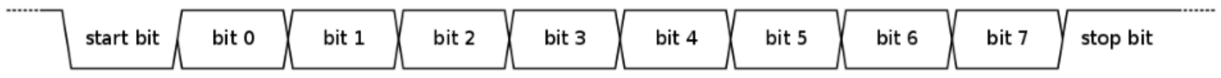


Figure 3.1: UART framing [16]. It has a start bit coded as a transition between a logic 1 and 0 and a stop bit coded as 1

3.5 Ethernet protocol

Ethernet is a family of computer networking. It can be used into three main categories of a network: Local Access Network (LAN), Metropolitan Area Network (MAN) and Wide Area Network (WAN). For this project, the system involves only LAN. The Ethernet Frame is the payload transported by the data unit of the Ethernet Protocol and it belongs to the Layer 2 of the OSI Model. As described by the standard, a standard Ethernet Frame is composed of the following fields; in Figure 3.2 the Ethernet Frame is shown. The Ethernet protocol refers to the Ethernet 2.0 version. [6]

3.5.1 Preamble

It is composed of seven bytes and in this system; they all correspond to the hexadecimal number 0x55.

3.5.2 Start of Frame Delimiter (SFD)

This byte indicates the beginning of the frame and is represented by the hexadecimal number 0xD5.

3.5.3 Destination Address

It is expressed as the Media Access Control address of the destination object of the message transported by the Ethernet Frame and it is composed by six bytes. The MAC address is a unique identifier whose code is not shared by any other object in the world if universally addressed. In this way of addressing, the MAC number is given according to the standard declared by IEEE. The MAC address can also be locally addressed by setting some bits belonging to the Most Significant Byte according to Big Endian format[17].

3.5.4 Source Address

As well as the destination address, it is composed of six bytes which compose the MAC Address of the device that is sending the message.

3.5.5 Ethernet Type

It is composed by two bytes and it is used by the destination device to understand the protocol used in the payload. In this system, the Ethernet Frame transports Internet Protocol Version 4 (IPv4) frames so the two bytes will always be set as the hexadecimal number 0x0800. The message can also be ARP; in this case, the code is 0x0806. [18]

3.5.6 Payload

The standard length of an Ethernet Frame’s payload is between 48 and 1500 bytes. Since the system is developed using the OSI model it will be a Layer 3 protocol, specifically IPv4[19] frames.

3.5.7 Cyclic Redundant Check (CRC)

Composed by four bytes, it is an error detecting code based on the computation of a signature of the frame by exploiting a polynomial division of all the content in the frame: addresses, Ethernet type, and payload. The CRC signature is calculated again when the message is received by the destination device and if it is not equal to the one in the frame it means that there was a failure in transmitting the message and some data were lost or modified. The CRC used in this system is CRC32. The CRC is not an error correction code, it only detects whether a fault in the message has occurred.

Preamble	SFD	Destination Address	Source Address	Ethernet Type	Payload	CRC
7	1	6	6	2	48-1500	4

Figure 3.2: Ethernet frame format, the numbers in the Figure indicate the number of bytes allowed by the standard. It is composed by a preamble, an SFD byte, 12 bytes for destination and source MAC address, the Ethernet type, the payload and four final bytes of CRC32

3.6 Point to Point Protocol (PPP)

The Point to Point Protocol is an OSI Layer 2 protocol used to link two nodes in a network without any other host or interfacing device during the communication. As described by the protocol, a standard PPP frame, as represented in Figure 3.3 is composed as follows. [7]

3.6.1 Flag

This byte is the start of the frame and it is the hexadecimal number 0x7E.

3.6.2 Address

This field is made by one byte and it is usually set to broadcast so the hexadecimal number 0xFF.

3.6.3 Control

The control byte is set to the hexadecimal number 0x03.

3.6.4 Protocol

These two bytes identify the type of protocol of the transported message; in this case, since it is IP it will be coded as 0x0021. Since the message can also be ARP and there is no indication from the standard for this type of message, it has been chosen the code 0x0081 since it is marked as unassigned from the standard itself.[20]

3.6.5 Information

This field has a variable number of bytes and it is the transported message.

3.6.6 CRC

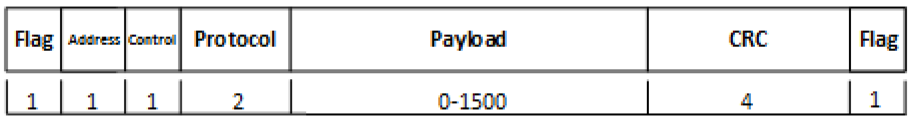
As in the Ethernet Frame case, these are four bytes of CRC32.

3.6.7 Flag

The hexadecimal number 0x7E is sent as the conclusive byte of the frame.

3.6.8 Escaping sequence

If the 0x7E byte is inside the payload, the protocol requires to translate that byte to the byte 0x7D into the new PPP frame followed by the XOR of the bytes 0x7E and 0x20. If the byte 0x7D is present in the payload, it is meant to be translated into 0x7D followed by the XOR between 0x7D and 0x20 bytes.



Flag	Address	Control	Protocol	Payload	CRC	Flag
1	1	1	2	0-1500	4	1

Figure 3.3: Point to Point frame format, the numbers in the Figure indicate the number of bytes allowed by the standard. It is started and ended by the Flag, it contains also the Address and Control fields, two Protocol field that described which protocol is the Payload message and four bytes of CRC32

3.7 Serial Line Interface Protocol (SLIP)

The Serial Line Interface Protocol is a standard used to encapsulate frames before being serialized. This protocol modifies the frame as follows.[8]

3.7.1 Frame END

This byte corresponds to the hexadecimal number 0xC0 and it is added at the beginning and at the end of the frame.

3.7.2 Frame ESC

In case the byte C0 is already inside the frame to convert, it is converted into two bytes 0xDB (the ESC byte) and 0xDC. If the Frame ESC is detected as well as a byte in the frame, it will be translated into the two bytes 0xDB and 0xDD.

3.8 Address Resolution Protocol (ARP)

The Address Resolution Protocol is a protocol used from the server device of the system to map all the MAC addresses of the clients connected to it[21]. The ARP protocol is based on a request in which the message is broadcast, and the content of the message is: *the IP address XX.XX.XX.Y asks who is XX.XX.XX.Z?* The answer to this message is unicast since there is no need for it to be broadcasted to all the devices. For purpose of simplicity, in the system that will be developed the answer will always be broadcast. Moreover, for this thesis, there is no need to describe the full details of the frame of the ARP.

3.9 Packet Internet Grouper (Ping) utility

Ping is a utility used to test the reachability of a host on Internet Protocol (IP) network [22]. IP is a protocol that works in the OSI Layer 3 [19]. When the Ping instruction is sent from a processor toward a specific IP address, the sender waits for the answer of the IP addressed device. If the answer is received it means that there is a connection between the devices. The utility allows also specifying in the command line extra field to monitor different features regarding the communication. The ping utility is used to test the full architecture inside the FPGA since in the Implementation phase two boards are connected through the IOChannel. When the Ping instruction is sent to a specific IP address, the sender waits for the answer of the IP addressed device. In this way, the full duplex communication can be tested.

4. The system inside the FPGA: how it was and how it will be

The interface that has to be developed is going to be designed and implemented in VHDL and uploaded on an FPGA. In this device, the OSI Layer 2 is implemented while the higher layers are managed outside of it. The new channels, which are going to be called IOChannels, are already printed on the PCB and a pin has already been selected for them in the FPGA

4.1 The system and purpose of the FPGA

The FPGA is a Switch. In telecommunication standard, a Switch is a device that drives incoming messages from one or more possible input and directs the messages to the proper output[4]. The messages can be broadcast, so sent to every other port connected to the switch, or unicast, so with a specified destination address and sent to a specific output port. All the components belonging to the FPGA are divided into two parts: *inbound* and *outbound*. When the message incoming toward the FPGA it is defined as *inbound* when it is going outside of it is defined as *outbound*. Both parts are independent on one another since the system is always full duplex. The desired system is represented in Figure 4.1.

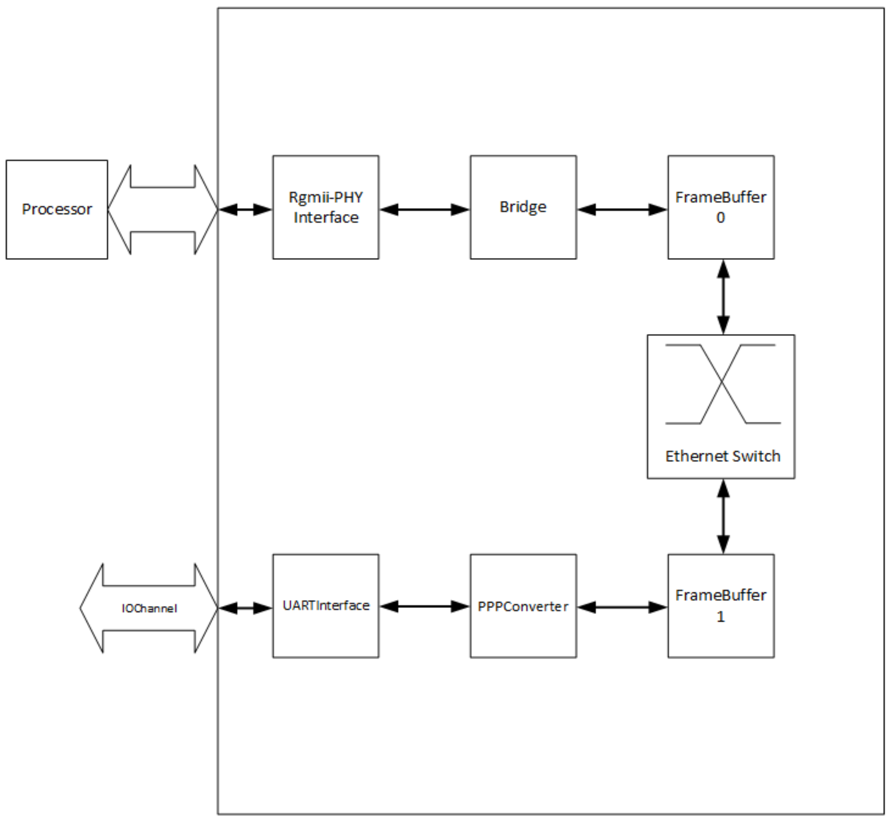


Figure 4.1: The communication system inside the FPGA. The message is sent by the Processor, it is sent to the IOChannel through the following components: Rgmii-PHY, Bridge, FrameBuffer, EthernetSwitch, FrameBuffer, PPPConverter, and UARTInterface

4.2 The flow of the message

Concerning all the used blocks, Figure 4.1 can be observed as a reference about how they are connected.

4.2.1 From the processor to the IOChannel

The message is sent by a processor and it is an Ethernet 2.0 Frame. The message encapsulated inside the Ethernet Frame can be an ARP request or an IP message. When the message enters the FPGA, it is translated from the Rgmii-PHY interface to an Ethernet Frame organized in bytes and it is saved into the FrameBuffer connected to the Port 0 of the Ethernet Switch. The Switch then decides which is the destination port and forwards the messages to it. From the destination Port the message is saved into another FrameBuffer and when a full frame is saved into it the PPPConverter starts to convert the message into PPP frame and forward it to the block UARTInterface. This block contains two core generated FIFOs (inbound and outbound) and a UART module that is responsible for the creation of the serial communication. The serial stream of bits then enters the LVDS buffer, proprietary of the FPGA, and the two LVDS signals exit the FPGA feeding one of the IOChannels. The message that comes from the processor enters the inbound part of all the components in the chain until it reaches the EthernetSwitch. When it is sent to another port the message is meant to be sent outside the FPGA, so it will be processed in the outbound part of the components.

4.2.2 From the IOChannel to the processor

When the message comes from the IOChannel, it is deserialized by the UARTInterface and stored in bytes in the inbound FIFO. The PPPConverter then acquires each byte and from them recreates the EthernetFrame using as destination address the MAC address of the processor. The Ethernet Frame is then saved into the FrameBuffer connected to one of the Ports of the Ethernet Switch. The Switch then redirects the message toward the proper Port, where the message is saved into the FrameBuffer. When the full frame is saved, the message is translated into PHY and sent to the processor.

4.3 Choice of the protocol on the IOChannels

These new channels are full duplex and LVDS, which means that for each channel there will be four serial wires: two in transmission (positive and negative) and two in reception (positive and negative). To have the fastest communication possible, the overhead of the frame, the ratio between the length of the non-payload bytes and the total length of the frame, must be the minimum to save time. For this reason, some standard protocols solutions have been considered to optimize the overhead due to the Ethernet Frame itself. As shown in Table 4.1 the SLIP protocol appears to be the best trade-off in terms of overhead proportion since it has only two more extra bytes to be added to the payload and, in case of full sized-frame, it has less overhead than PPP. The problem with SLIP is that it has no error detection field, as the CRC in the Ethernet and PPP frame. Moreover, the PPP is a very flexible protocol since it has the Address, Control, and Protocol fields while having less overhead with respect to the Ethernet Frame if there is no escaping needed. Flexibility is needed since the element that is going to receive the PPP frame is not yet designed. For all these reasons, the PPP was chosen as best protocol as a trade-off among performances, safety, and flexibility.

Table 4.1: Possible Protocol overhead scenarios

Type	Theoretical Overhead, Standard Payload length L bytes	Maximum Overhead, all characters must be converted (SLIP and PPP)
Ethernet Frame	18/L	18/L
SLIP	2/L	3000/L
PPP	10/L	30000/L

4.4 Involved components

Many different components are involved in the flux of the messages. In the following both the signal interface and the timing, when relevant, are shown. All the signals not mentioned in the description of the components are not to be considered since not relevant to the system.

4.4.1 Reduced Gigabit Media-Independent Interface (Rgmii) - PHY Interface

This component allows converting the Ethernet frame from the format used into the OSI Layer 2 into the one used in the OSI Layer 1, the physical layer, and vice versa. This block acts as a synchronizer and helps the processor to interface the FPGA. There are three main signals from the processor: data on four bits, the enable signal and the clock which might be different with respect to the one used internally in the FPGA. These signals are used according to the Rgmii standard.[23]

The signal interface was studied to create the complete testbench of the system. In Figure 4.2 it is possible to see the input and output signals belonging to this component. The clock used for the interface toward the processor is not necessarily the one in the system since it can be different. Inside the component, there are specific processes to synchronize the signals. The data are received in frames composed by four bits and then merged so that they are sent as bytes inside the system of the FPGA. When a data is valid to be read, the processor asserts the CTL signal. The 4-bits frames are incoming toward the FPGA coded as little-endian: this component creates the bytes as big-endian so that the message can be correctly managed inside the FPGA that uses this order. The inbound part of the communication, which is the one just described, can be noticed in Figure 4.3. In the outbound part, the component takes the bytes and separates them in four bits data, synchronize the clock with the one received from the processor and asserts the enable signal CTL when the data are valid.

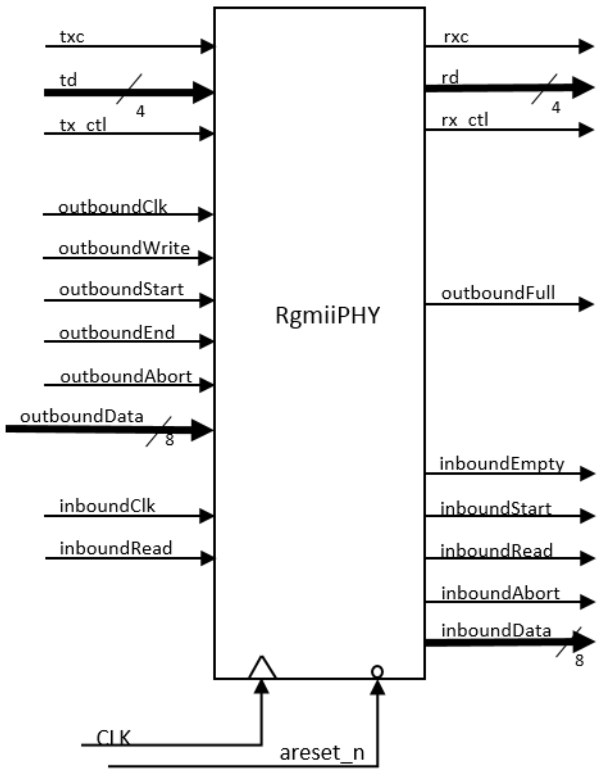


Figure 4.2: Rgmii-Phy signals interface

The timing diagram showing the writing action from the processor toward the Rgmii-PHY features three signals: the clock, the CTL and the data. The CTL signal acts as a data valid signal as the Rgmii standard prescribes (Figure 4.3). The Inbound interface between the Rgmii-PHY component and the Bridge is shown in Figure 4.4. The read signal is asserted every two clock cycles and when the bridge has asserted the start signal the sequence can be read. If the signal end is strobed the frame is finished, and the Bridge will wait for another start strobe to read. As for the outbound part, the interface is the same with start and end strobes, the write signal behaves as the read but there is no delay between data and the assertion of the write signal.

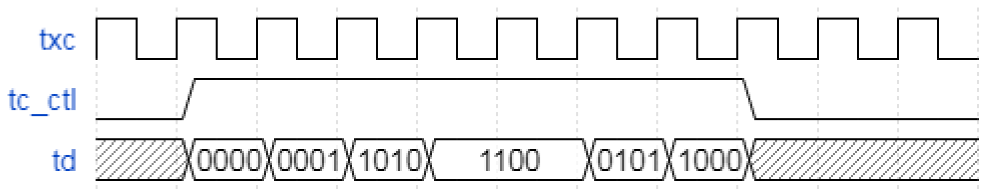


Figure 4.3: Timing diagram of the signals used by the processor to send a message to the FPGA. The interface component is Rgmii-PHY. If the ctl signal is asserted the data is valid. The data are an example and are not indicative of a proper Ethernet frame.

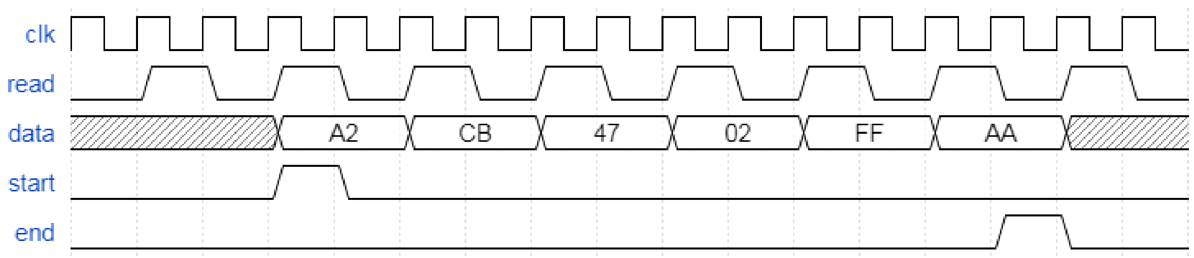


Figure 4.4: Timing diagram of the interface between the Rgmii-PHY and Bridge components. Inbound Interface. The start and end signal delimit the frame. The data are an example and are not indicative of a proper Ethernet frame.

4.4.2 Bridge Interface

A network bridge is a computer networking device that fragments the network[24]. It receives a message and decides if forward it or discard it. It is simpler than a Switch since the communication between two blocks is not independent, but it is chosen by the Bridge itself. In this system, the Bridge decides if the inbound information can go toward the EthernetSwitch or other parts of the FPGA which will be ignored since not useful for this project. In the inbound part of the Bridge, the Preamble and the SFD of the Ethernet Frame are removed while in the outbound part it is added again. These parts of the Ethernet frame are also checked and if they do not have the expected value they are discarded. In Figure 4.5 the bridge signals interface is shown. With this component, the timing will not be described since it is already shown for the two components to which it is connected to, the Rmgii-PHY and the FrameBuffer.

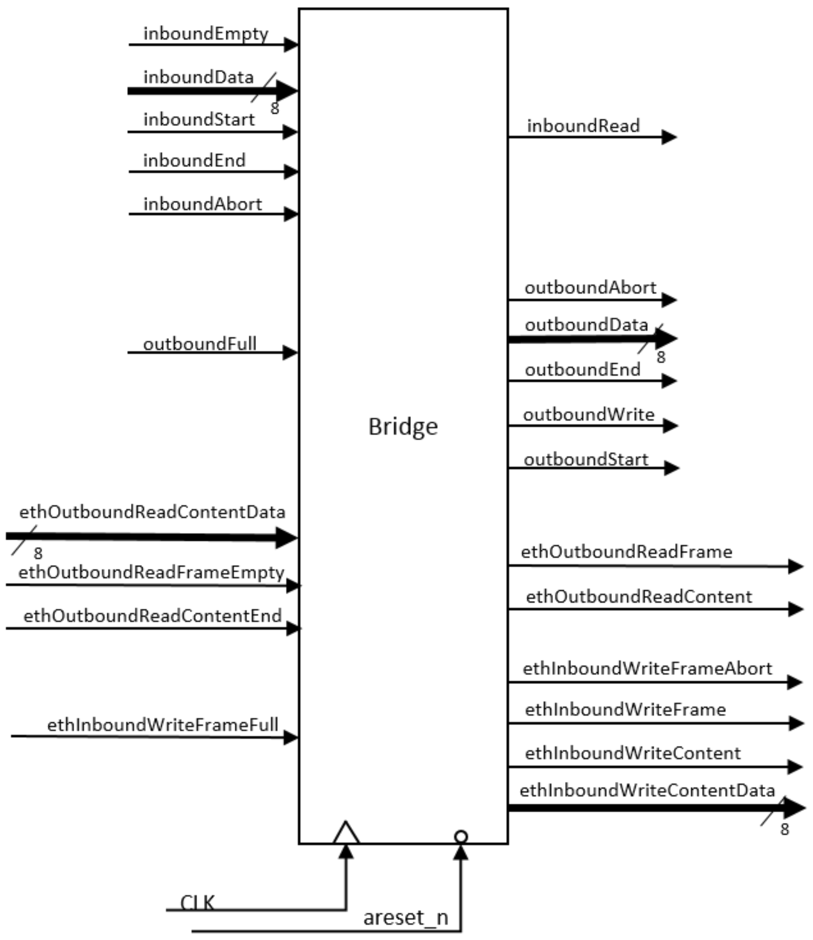


Figure 4.5: Bridge signals interface

4.4.3 FrameBuffer

To synchronize the flow of communication, each Port is connected to a FrameBuffer (Figure 4.6). This buffer can store up to 16 KB, that are 10 full-size Ethernet Frames, and it is divided as well in the Inbound and Outbound part.

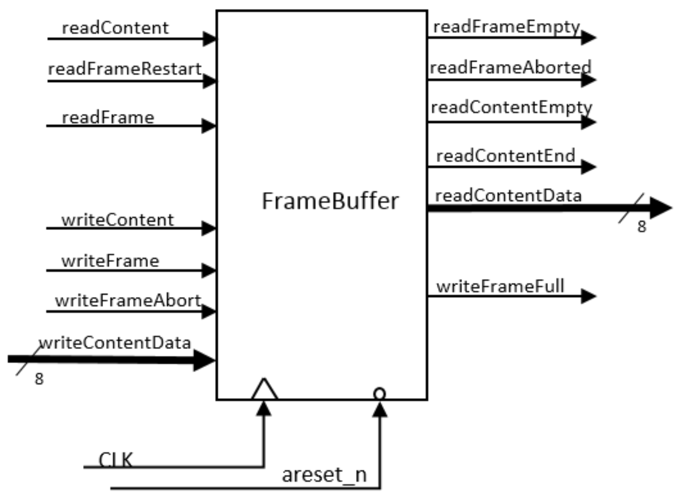


Figure 4.6: FrameBuffer signals interface. Since the inbound and outbound parts have the same interface, the signals shown refer to a generic part.

The read operation is performed by asserting the signal ReadContent. One clock cycle later on the signal ReadContentData the bytes will start to appear. When a frame is completely read, the buffer asserts the signal ReadContentEnd for one clock cycle. If the user wants to read the next frame, the signal ReadFrame has to be asserted for one clock cycle as a strobe while the signal ReadContentData is not asserted (Figure 4.7). The write operation is executed by asserting the signal WriteContent and in the same clock cycle starting to give the frame as input on the signal WriteContentData. When the user wants to write a new frame, the signal WriteFrame has to be asserted for one clock cycle as a strobe (Figure 4.8).

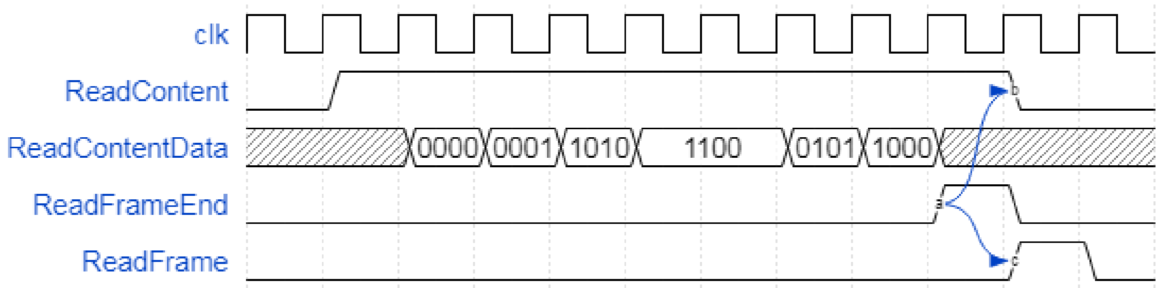


Figure 4.7: Timing diagram of the reading operation on the FrameBuffer component. The ReadContent signal is asserted until the ReadFrameEnd is asserted and the frame is finished. To read the next frame the ReadFrame signal must be strobed for one clock cycle. The data are random values.

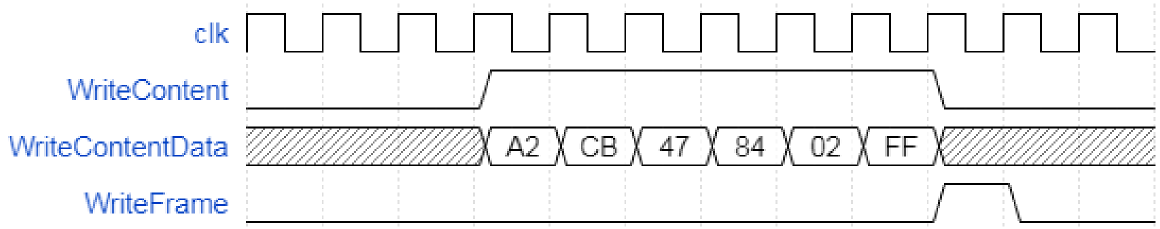


Figure 4.8: Timing diagram of the writing operation on the FrameBuffer component. While the WriteContent signal is asserted, the data are written inside the buffer. When a new frame has to be written, the WriteFrame signal is strobed for one clock cycle.

4.4.4 Ethernet Switch

The main component inside the FPGA that acts as a switch is the EthernetSwitch: it is a block developed by Bombardier that receives the incoming signal, decide which other object connected to it is the destination and forward the message to it. Each device that needs to exchange messages to the other devices connected to the FPGA is connected to a Port on this component. All the incoming and outcoming messages are Ethernet Frames, so the addressing of the destination and source is based on the MAC Address specified in the message. The peculiarities of this block are that it is self-learning and the decision of which port can talk is taken by a WishBoneCrossbar component [25] based on a Round-Robin Arbiter [26], so each port has the same priority to talk. All the signals incoming toward the EthernetSwitch belongs to the inbound part of the system. Therefore, all the outcoming messages belong to the Outbound part of the system. The signals belonging to the interface are used in the same way as the ones of the FrameBuffer, as well as the timing. The only difference is that each signal of the FrameBuffer is declared as a vector of length equal to the number of ports instead of a single bit control signal.

4.4.5 UARTInterface: core generated FIFOs and UART module

This component contains three components: two identical Xilinx Core-generated FIFO and a UART module. The two FIFOs are one for the inbound e one for the outbound part. The interface toward the PPPConverter is the same of one FIFOBuffer (Figure 4.9) and it makes possible to write on the outbound FIFO and read from the inbound FIFO. The UART module is interfaced inside the UARTInterface component with the two FIFOs and the only signals that interfaces the other main components are the IOSignal_i and IOSignal_o which are respectively the input and output serial line connected to one of the physical IOChannel. Moreover, this block is the only one of the involved components that has a positive reset.

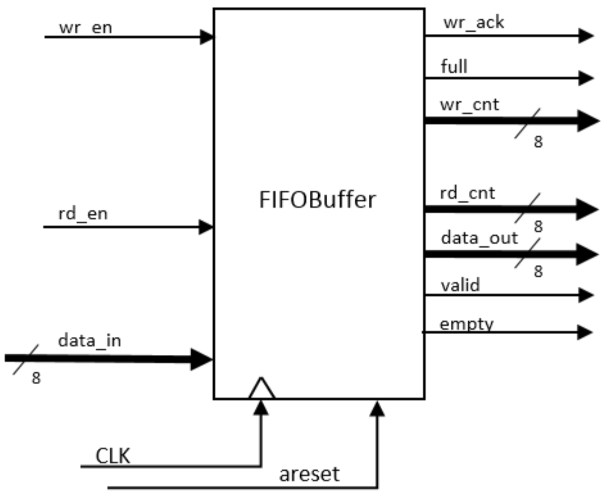


Figure 4.9: FIFOBuffer signals interface

The timing diagrams that are relevant to the project are the Core generated FIFOs interface [27]. Since they are identical in the inbound and outbound part, the indicated signals will be named in a general way. The signal rd_en is asserted and one clock cycle later the data can be read on the pin data_out. The data are valid to be used if the signal valid is also asserted by the FIFO. The signal empty notifies if there are data in the FIFO by becoming 0 (Figure 4.10). Concerning the writing operation, in the same clock cycle, the wr_en signal has to be asserted and the data must be present on the data_in pin. When the FIFO has correctly written the data, it asserts the signal wr_ack one clock cycle after the data has been sampled by the FIFO (Figure 4.11).

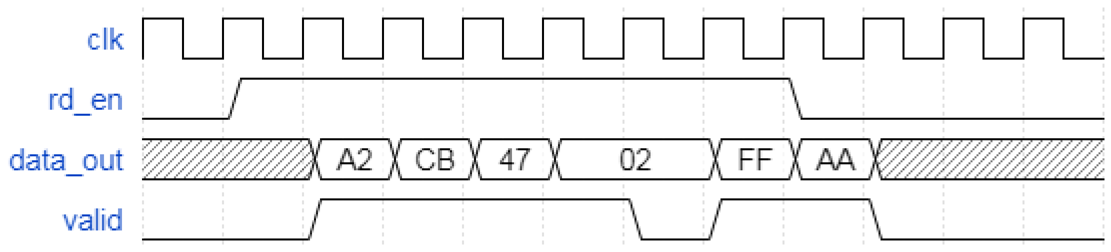


Figure 4.10: Timing diagram of the reading operation from the FIFOBuffer. The rd_en signal is asserted and one clock cycle later the data are read. The data is the correct one only if the valid signal is asserted as well. The data are random values.

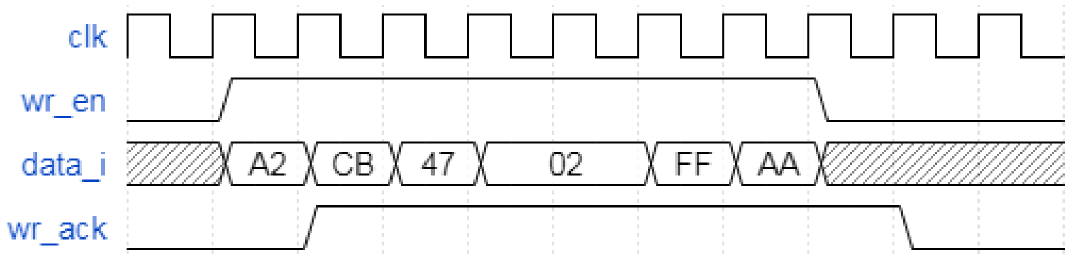


Figure 4.11: Timing diagram of the writing sequence on the FIFOBuffer component. The wr_en signal is asserted and the data is written into the FIFO. If the memory has correctly written the data, the wr_ack is asserted. The data are random.

4.4.6 PPPConverter

This component is the main object of the thesis project. Since all the communication is divided into inbound and outbound, the two parts will be described separately. The communication is full duplex, so they are meant to be always independent of each other.

5. Design of PPPConverter

The Converter is divided into two parts: Inbound and Outbound. They work in parallel and are not meant to communicate with each other. The approach used to design the VHDL code is the behavioral one, as used by all the other codes belonging to this system. Therefore, the Data Path scheme will not be introduced while the most relevant importance is given to the Control Flow Chart. The two protocols have different length and values in the head and tail of the frame so if the system must be designed to have the least latency possible, for this reason in both parts the writing of the new head and the reading of the incoming message will be done in parallel. In both cases, a Data Flow management graph will show how this procedure is managed during the evolution of the states. Since all the signals names have as radix the name of the part to which they belong to, inbound and outbound, while describing the two different part the radix will be neglected and it must be considered as the name of the part that is described.

5.1 Error and escaping management

Since the FPGA works in a high safety environment, some studies about error detection and correction inside the FPGA and the system must be considered. As far as the system in which the FPGA works is concerned, when a frame is faulty is not a responsibility of the FPGA to correct the mistake: if an error is detected the frame is discarded and the processor that has sent the message will send it again until it receives an answer. The Safety Layer of the network system is the one responsible for the management of these faulty situations. For this reason, the signal wr_ack of the FIFO is not checked in the PPPConverter component: if the UART receive a faulty byte the inbound part of the PPPConverter on the other side of the channel will detect the problem and discard the faulty message. To discard a message, the FrameBuffer has a dedicated signal called WriteFrameAbort: if the signal is asserted the buffer will not save the rest of the frame and will delete the bytes already written inside of it.

The escaping feature required by the PPP protocol standard declaration was not implemented in the PPPConverter component. This decision was taken to have a first version of the converter that could be simple enough to be implemented and at the same time could introduce the PPP frame to the system. This was also possible since the two boards that will be used for the implementation test (see Chapter 7) do not use the special character 0x7E in their framing. This aspect was verified [28] [29] using the tcpdump command on the processor interface toward the FPGA: the frames that are sent in both the ARP and Ping messages never contain the Flag. The escaping feature must be added in the future implementations of the component in order to make the communication fully standard.

5.2 ShiftMemory

In both parts, the length of the frame is not known but there is a signal or data value that indicates when the frame ends. Moreover, the CRC32 signature calculated in both the protocols is going to be different because the head of the frame is different, so the last four bytes have to be discarded and substituted with the new signature. In the Inbound part, the end of frame delimiter is the byte 0x7E: when it is read, the frame is finishes and the previous four bytes are the old CRC32. In the Outbound part, when the signal ReadFrameEnd is asserted, it means that the four last bytes are the old CRC32. Since there is the same behavior in both parts, the ShiftMemory component, designed to create the proper tail to the frame, will be described as one component which is independent on the part of the PPPConverter in which works. The only difference is in the reset value of the output of this component: in the inbound part will be set at zero while in the outbound will be set at 7E since the PPP protocol prescribes that value to be the idle state of the line. The ShiftMemory component is composed of four eight-bit signals that, when the signal enable is asserted, shift their value to the next one. When the end of frame moment is reached, the last value to be read is the output of the ShiftMemory and the content of it, the old CRC32, is discarded. This means that all the payload bytes are shifted into this component before being written into the FrameBuffer or the FIFO and therefore delayed by four clock cycles (Figure 5.1). This choice raises the latency of the frame, but the overhead of four clock cycles compared to the latency given from the overall system is dispensable. Since the component is described using the behavioral VHDL, it is not possible to draw an accurate Data Path.

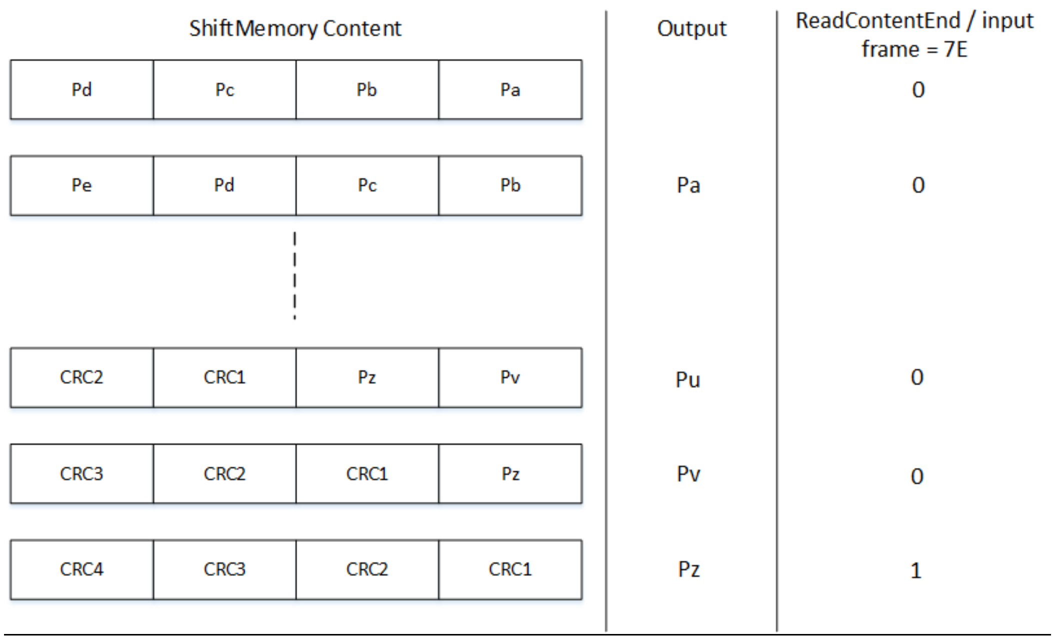


Figure 5.1: Behavior of the ShiftMemory component. When the end of frame trigger is asserted, the last shifted out content is the data and the content of the ShiftMemory is the old CRC, that can be discarded.

5.2.1 Pseudocode

The behavior of the ShiftMemory component is explained in the following pseudocode. Please note that after every semicolon a new clock cycle will occur, as it happens for the Finite State Machine in the VHDL code.

```
If (reset = 0) data1, data2, data3, data4 <= 0; // if inbound,
 // if outbound 7E
If (enable = 1)
{    data1 <= data_in,
     data2 <= data1,
     data3 <= data2,
     data4 <= data3,
     data_out <= data4;
}
```

5.2.2 Timing diagram

The timing diagram act according to the described behavior. When the shiftEnable signal is asserted the content of the four registers is shifted toward the next register in line otherwise the content remains as it is. The component is synchronous, so this behavior only happens on a positive edge of the clock signal. If the shiftReset signal is asserted, the value of all the internal signals is 0x00 for the inbound and 0x7E for the outbound part (Figure 5.2).

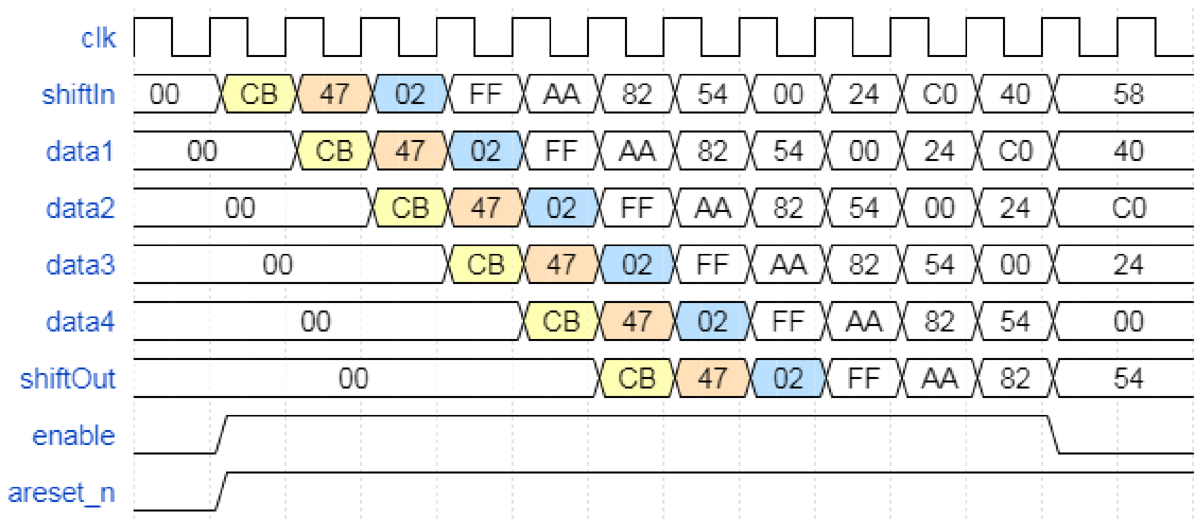


Figure 5.2: Timing diagram of the ShiftMemory. When the enable is asserted the bytes are shifted creating the diagonal pattern. When the reset is asserted the component assumes the 0x00 value if inbound and 0x7E if outbound.

5.3 Calculation of the CRC32 signature

In both Ethernet and PPP frame, a CRC32 signature is needed. The input of the component, developed by Bombardier Transportation, is composed by an 8-bit signal which is the byte that has to be added to the signature and a 32-bit signal that is reset at the value FF and when a new byte is processed it has to be set at the 32-bit output signal of the CRC32 itself. This component is asynchronous. This procedure is used in both inbound and outbound parts as shown in the Pseudocodes.

5.4 Inbound

The Inbound part is the one responsible for converting the message from PPP to the Ethernet Frame. It should read the frame, check whether the frame has the correct five bytes of the preamble of the PPP protocol while writing on the FrameBuffer the destination and source MAC address and the Ethernet Type. Then it is going to transfer the payload and finally the new calculated CRC32 bytes. The CRC32 is the signature generated by the two MAC addresses, the Ethernet Type and the payload bytes. Since the length of the frame is not fixed, all the data that will be written into the buffer are delayed by a shift register called ShiftMemory which is going to flush the elements remaining inside the shift register when the Flag byte is read, acting as an end frame. In this way, there is no need to count the length of the total frame. Finally, since there are two possible types of payload, there will be a check to update the corresponding Ethernet Type field according to the Protocol field inside the PPP Frame. In case the Protocol is 0x 0021 the payload is IP so translated to 0x0800 if 0x0081 it is ARP so translated to 0x 0806, otherwise it is an error. The coding of the type of frame is decided by the standard.

5.4.1 Pseudocode

The behavior of the inbound part of the component is explained in the following pseudocode. Please note that after every semicolon a new clock cycle will occur, as it happens for the Finite State Machine in the VHDL code.

```
finished=0;
if (FifoEmpty=0) // data is available in the FIFO
{ while (! Finished) {
     if(bufferfull=0) output= destinationMAC (47 downto 40)
     updateCRC,
   else // wait for the buffer to be not full;
```

```
  if(bufferfull=0) output= destinationMAC (39 downto 32)
updateCRC,
  else // wait for the buffer to be not full;

  if(bufferfull=0) output= destinationMAC (31 downto 24)
updateCRC,
  else // wait for the buffer to be not full;

  if(bufferfull=0) output= destinationMAC (23 downto 16)
updateCRC,
                 read_byte
  else // wait for the buffer to be not full;

  if (bufferfull=0 & datavalid) output= destinationMAC (15 downto 8)
updateCRC,
                 read_byte
                 if input!= 7E error, reset_shift, finished=1
  else // wait for the buffer to be not full;

  if (bufferfull=0 & datavalid) output= destinationMAC (7 downto 0)
updateCRC,
                 read_byte
                 if input!= FF error, reset_shift, finished=1
  else // wait for the buffer to be not full;

  if (bufferfull=0 & datavalid) output= sourceMAC (47 downto 40)
updateCRC,
                 read_byte
                 if input!= 03 error, reset_shift, finished=1
  else // wait for the buffer to be not full;

  if (bufferfull=0 & datavalid) output= sourceMAC (39 downto 32)
updateCRC,
                 read_byte
                 if input!= 00 error, reset_shift,finished=1
  else // wait for the buffer to be not full;

   if (bufferfull=0 & datavalid)
       if input = 21 ethType =00
       else if input = 81 ethType = 06
       else error, reset_shift, finished = 1;
       output= sourceMAC (31 downto 24)
   updateCRC,
                 read_byte
  else // wait for the buffer to be not full;

   if (bufferfull=0 & datavalid) output= sourceMAC (23 downto 16)
   updateCRC,
           read_byte
           shiftMemory= input; // start to shift in the payload
  else // wait for the buffer to be not full;

       if (bufferfull=0 & datavalid) output= sourceMAC (15 downto 8)
   updateCRC,
           read_byte
           shiftMemory= input; // shift in the payload
  else // wait for the buffer to be not full;

       if (bufferfull=0 & datavalid) output= sourceMAC (7 downto 0)
   updateCRC,
           read_byte
           shiftMemory= input; // shift in the payload
  else // wait for the buffer to be not full;

       if (bufferfull=0 & datavalid) output= x”08” // shift EthType
   updateCRC,
           read_byte
           shiftMemory= input; // shift in the payload
  else // wait for the buffer to be not full;

       if (bufferfull=0 & datavalid) output= ethType // shift EthType
   updateCRC,
           read_byte
           shiftMemory= input; // shift in the payload
  else // wait for the buffer to be not full;

        while (input!= 7E)
        if (bufferfull=0 & datavalid) output= shiftOut
             updateCRC,
                   read_byte
                  shiftMemory= input;
           else // wait for the buffer to be not full;

        if(bufferfull=0) output= crc(31 downto 24)
   else // wait for the buffer to be not full;

        if(bufferfull=0) output= crc(23 downto 16)
   else // wait for the buffer to be not full;

        if(bufferfull=0) output= crc( 15 downto 8)
   else // wait for the buffer to be not full;

        if(bufferfull=0) output= crc(7 downto 0)
   else // wait for the buffer to be not full;

        // packet translated correctly, update strobe for input buffer
   update_strobe_buffer, finished = 1;
   }

        }
```

5.4.2 Data Flow Management

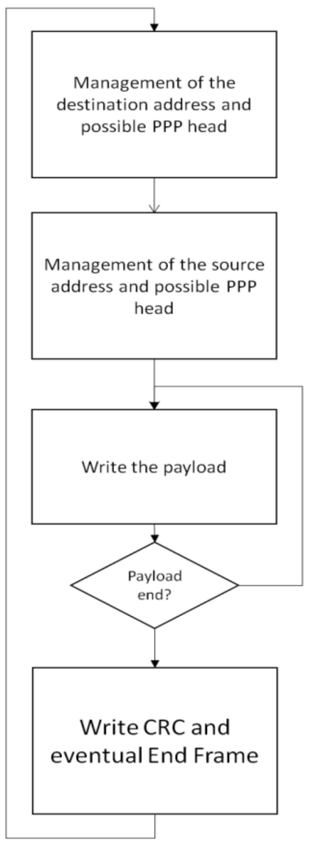
The Data Flow management of the Ethernet Frame to be written and the PPP frame to be converted was studied to have the fastest conversion possible while checking if the received frame was a correct PPP frame (Figure 5.3). In total there are 22 states, in sequence order they are: one IDLE state, 6 to write the destination MAC address, 6 for the source address, 2 for the Ethernet Type field, one to write the payload until the frame is not finished, 4 for writing the CRC32 signature and the last one to update the WriteFrame strobe for the FramBuffer. By exploiting the ShiftMemory component, while the MAC addresses are directly written into the FrameBuffer, the reading of the PPP frame can start. Since the ShiftMemory takes 5 clock cycles to bring on its output one byte, the reading and shifting of the payload of the PPP frame can start 5 states before the STATE_RD_WR_PAYLOAD in which the output of the ShiftMemory is written into the FrameBuffer. This means that the first reading of the payload has to start in the state in which the fourth byte of the source address is written. Since the read signal must be asserted one clock cycle before the data, it will be asserted in the state in which the third byte of the source address is written. Moreover, since the FSM is written in one process, the writing action must be asserted one clock cycle before, so the state in which the second byte of the Ethernet Type is written. Moreover, the PPP frame has 5 bytes of preamble before sending the payload. This means that the effective reading of the FIFO can start during the writing of the fifth byte of the destination address and the read enable signal must be asserted in the previous state, during the writing of the fourth destination byte. During the reading of the head of the PPP frame, if one of the bytes has a different value, the FSM goes into the STATE_ERROR in which the WriteFrameAbort signal is asserted, the shift register is reset, and the FSM goes back to the idle state. If not, when the value 0x7E is read again, the content of the ShiftMemory is deleted and in the next four states, the four bytes of the CRC32 will be written directly into the FrameBuffer. A final state updates the strobe signal for the FrameBuffer and a new frame can be converted. To summarize, the reading is always performed from the inboundFIFO component, the writing is done in two different ways: the MAC addresses, Ethernet Type and CRC32 signature are written directly into the FrameBuffer, the payload is shifted into the ShiftMemory and the output of this component is written into the FrameBuffer delayed by 4 clock cycles. Before every writing operation, the FrameBuffer signal WriteFrameFull has to be checked: if it is asserted the FSM has to stay in the same state in which it is and wait for the buffer to have space again. The same happens when the data from the FIFO is not marked as valid by the homonym signal: if it is not asserted the FSM has to wait for a new valid signal.

State of FSM	Content read	Content of ShiftMemory	ShiftMemory output	Byte wrote in the FIFO
IDLE	0	0 0 0 0	0	X
WR_DESTINATION_1	0	0 0 0 0	0	X
WR_DESTINATION_2	0	0 0 0 0	0	X
WR_DESTINATION_3	0	0 0 0 0	0	X
WR_DESTINATION_4	0	0 0 0 0	0	X
WR_DESTINATION_5_RD_FLAG	7E	0 0 0 0	0	X
WR_DESTINATION_6_RD_ADDRESS	FF	0 0 0 0	0	X
WR_SOURCE_1_RD_CONTROL	03	0 0 0 0	0	X
WR_SOURCE_2_RD_PROTOCOL_1	00	0 0 0 0	0	X
WR_SOURCE_3_RD_PROTOCOL_2	21	0 0 0 0	0	X
WR_SOURCE_4_RD_PAYLOAD	PA	PA 0 0 0	0	X
WR_SOURCE_5_RD_PAYLOAD	PB	PB PA 0 0	0	X
WR_SOURCE_6_RD_PAYLOAD	PC	PC PB PA 0	0	X
WR_ETHTYPE_1_RD_PAYLOAD	PD	PD PC PB PA	0	X
WR_ETHTYPE_2_RD_PAYLOAD	PE	PE PD PC PB	PA	X
RD_WR_PAYLOAD	...	PF PE PD PC	PB	PA
RD_WR_PAYLOAD	PZ	C'4 C'3 C'2 C'1	PZ	X
WR_CRC_1	X	C'4 C'3 C'2 C'1	PZ	C1
WR_CRC_2	X	C'4 C'3 C'2 C'1	PZ	C2
WR_CRC_3	X	C'4 C'3 C'2 C'1	PZ	C3
WR_CRC_4	X	C'4 C'3 C'2 C'1	PZ	C4
END_WR_STROBE	X	0 0 0 0	PZ	X

Figure 5.3: Data Flow management in the inbound part of the PPPConverter component. The read content, the content of the ShiftMemory, its output and the byte written in output are shown for each state of the FSM.

5.4.3 Control flow chart

Since the flow is composed of many states, to make the Finite State Machine more understandable, it has been divided into four parts (Figure 5.4). The Control Flow charts describe the evolution of the states in the Inbound FSM. They follow the same algorithm of the Pseudocode and the Data Flow described before. They contain for each state the signals that must be asserted or not asserted according to the given timing diagrams and the behavior previously described. They can be found in the Appendix Section 14.1. The designed FSM has both Moore and Mealy states: the condition of the buffer to be empty or full has always to be checked before a respectively reading or writing operation which leads to reading the input and not only the output of the FSM generating the need to use also Mealy states. In the Control Flow Chart, the Moore states are indicated by squared states while the Mealy ones are in a more rounded shape.



Management of the destination address and possible PPP head
Management of the source address and possible PPP head
Write the payload
Payload end?
Write CRC and eventual End Frame

Figure 5.4: General division of the control flow chart. For sake of simplicity, four parts have been identified. The destination MAC address management, the source MAC address and possible PPP header, the payload, and the CRC and end-frame.

5.4.4 Timing diagram

The timing diagrams have been separated into four different parts using the same approach as the Control Flow Diagrams. The four stages that are shown are the start of the writing phase of the Ethernet Frame, the start of the reading phase of the PPP frame, the payload reading and writing phase and the final states of the FSM where the system goes back to the IDLE state. The values of the data used in all the timing diagrams of this document are purely random and are not referred to anything specifically. The timing diagrams show only the significant signals for the described phase. Moreover, not all the states are shown since some have the same behavior, only a sample of them and the special ones will be shown.

When the FIFO is not empty, the PPPConverter starts to write the destination MAC address of the Ethernet frame: it writes directly in the FrameBuffer (Figure 4.8). In Figure 5.5 it is possible to see the evolution of the states from IDLE to WR_DESTINATION_1 and the involved control signals. In this Figure it is also shown the usage of the CRC32: it will not be shown again in the next timing diagrams. During the writing of the destination MAC address, the FSM starts to read the PPP frame. The rd_en signal is asserted one clock before the data and each byte of the PPP preamble that is received is checked: if it is not the expected one the system aborts the frame and restart from IDLE (Figure 5.6). When the FSM has finished reading the preamble of the PPP frame, it starts to read the payload (Figure 5.7). Finally, the FSM writes the CRC32 signature, update the signals for the FrameBuffer and restarts from the IDLE state (Figure 5.8).

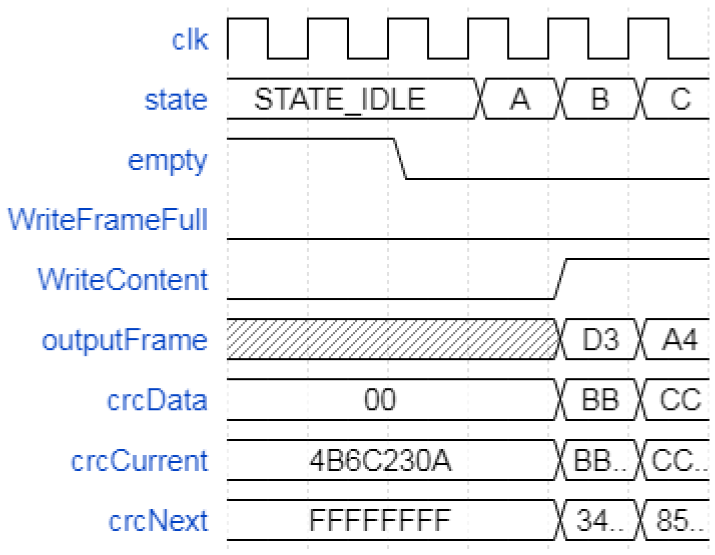


Figure 5.5: Start of the Inbound FSM, the system begins to write the destination MAC address into the FrameBuffer. The states have been coded to make the timing diagram more understandable. The corresponding states are shown in Table 5.1.

Table 5.1: Coding of the states for the timing diagram in Figure 5.4

State name	Coded name
STATE_WR_DESTINATION_1	A
STATE_WR_DESTINATION_2	B
STATE_WR_DESTINATION_3	C

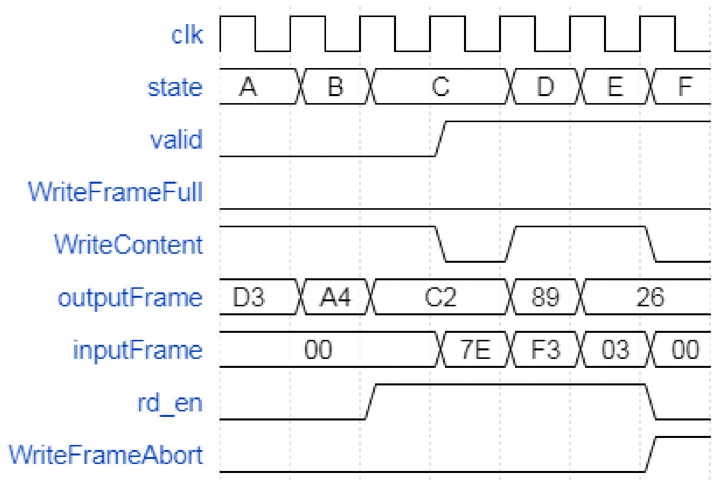


Figure 5.6: Timing diagram of the start of the reading part of the PPP frame, inbound part. A faulty byte is detected, and the system aborts the frame. The states have been coded to make the timing diagram more understandable. The corresponding states are shown in Table 5.2.

Table 5.2: Coding of the states for the timing diagram in Figure 5.6

State name	Coded name
STATE_WR_DESTINATION_3	A
STATE_WR_DESTINATION_4	B
STATE_WR_DESTINATION_5_RD_FLAG	C
STATE_WR_DESTINATION_6_RD_ADDRESS	D
STATE_ERROR	E
STATE_IDLE	F

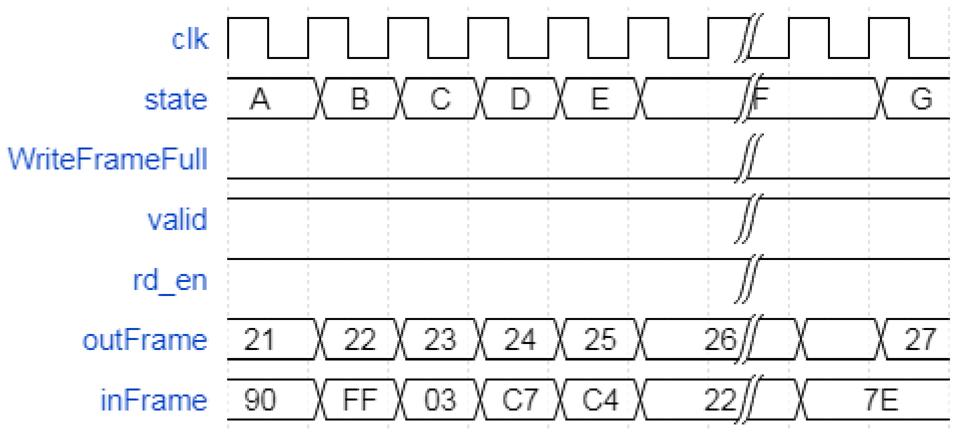


Figure 5.7: Timing diagram of the handling of the payload in the Inbound part of the PPPConverter component. The states have been coded to make the timing diagram more understandable. The corresponding states are shown in Table 5.3.

Table 5.3: Coding of the states for the timing diagram in Figure 5.7

State name	Coded name
STATE_WR_SOURCE_4_RD_PAYLOAD	A
STATE_WR_SOURCE_5_RD_PAYLOAD	B
STATE_WR_SOURCE_6_RD_PAYLOAD	C
STATE_WR_ETHTYPE_1_RD_PAYLOAD	D
STATE_WR_ETHTYPE_1_RD_WR_PAYLOAD	E
STATE_RD_WR_PAYLOAD	F
STATE_WR_CRC_1	G

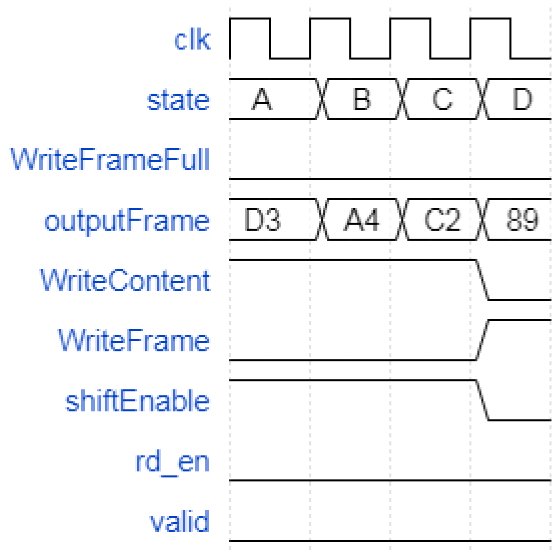


Figure 5.8: Timing diagram of the end of the evolution of the inbound FSM. The states have been coded to make the timing diagram more understandable. The corresponding states are shown in Table 5.5.

Table 5.4: Coding of the states for the timing diagram in Figure 5.8

State name	Coded name
STATE_WR_CRC_3	A
STATE_WR_CRC_3	B
STATE_END_WR_STROBE	C
STATE_IDLE	D

5.4.5 Source and destination MAC address management

In the inbound part, the MAC addresses must be rewritten into the new Ethernet Frame. To make the system simple, they are two values that are an input for the PPPConverter component. The destination address is set as the processor universal MAC address decided from the vendor. The source MAC address is decided by the VHDL code and it is a locally administered address decided by the designer. To set that address as locally administered, the second Least Significant Bit (LSB) of the first byte of the source address is set to 1. The way that decides the values of the two addresses is not the final version of the management of the addresses and it must be considered as a future implementation. To see a proposed future implementation, see Section 8. Future Implementations[17].

5.5 Outbound

The Outbound part is responsible for the translation from Ethernet to PPP of the frame. The component reads the destination and MAC address while writing into the OutboundFIFO the preamble bytes of the PPP protocol and updating the Protocol field according to the received Ethernet Type. In case a 0x0800 is received it will be translated as 0x0021, if 0x0806 is received it will become 0x0081. If something different is received it means that there was an error in the transmission and the frame will be discarded. Successively, the payload is written, then the four bytes of CRC32 and finally the Flag byte.

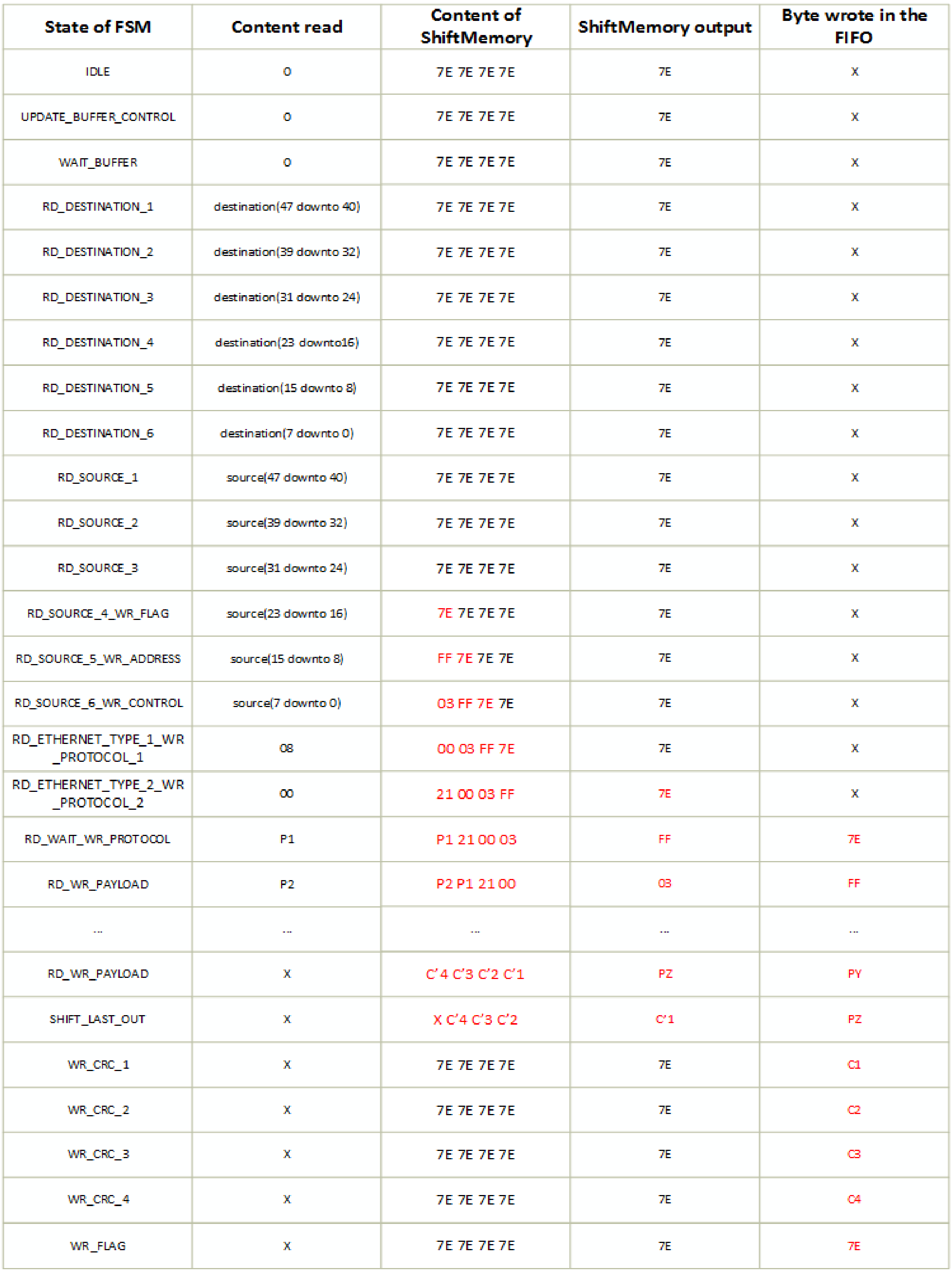
5.5.1 Pseudocode

The pseudocode that follows describes the expected behavior of the outbound part of the PPPConverter. It is not related to a specific code, but it is meant to give an insight of the followed algorithm.

```
finished=0;
if (bufferEmpty=0) // data is available in the buffer
{ while (finished = 0) {

     read_destinationMAC (47 downto 40);

     read_destinationMAC (39 downto 32);

     read_destinationMAC (31 downto 24);

     read_destinationMAC (23 downto 16);

     read_destinationMAC (15 downto 8);

     read_destinationMAC (7 downto 0);

     read_sourceMAC (47 downto 40);

     read_sourceMAC (39 downto 32);

     read_sourceMAC (31 downto 24)
     if FIFOfull=0 output= 7E, updateCRC, else; //wait

     read_sourceMAC (23 downto 16)
     if FIFOfull=0 output=FF, updateCRC, else; //wait

     read_sourceMAC (15 downto 8)
     if FIFOfull=0 output=03, updateCRC, else; //wait

     read_sourceMAC (7 downto 0)
     if FIFOfull=0 output=00, updateCRC, else; //wait

      read_Eth (15 downto 8)
      if input=00 output=21, updateCRC,
      elsif input = 06 output= 81, updateCRC,
      else error, reset_shift, finished=1;

      read_Eth (7 downto 0)
      shiftMemory=input, updateCRC;

      while(dataend=0) {
      if FIFOfull=0
      shiftMemory=input, write_output, updateCRC;} // write payload

      if FIFOfull=0
      shiftData, write_output, updateCRC; // shift last data out

      if FIFOfull=0
      writeoutput= CRC(31 downto 24);

      if FIFOfull=0
      writeoutput= CRC(23 downto 16);

      if FIFOfull=0
      writeoutput= CRC(15 downto 8);
      if FIFOfull=0
      writeoutput= CRC(7 downto 0);

      if FIFOfull=0
      writeoutput= 7E, finished=1; // write flag and start again

}

}
```

5.5.2 Data Flow Management

The Data Flow management in the Outbound process was studied to have the fastest conversion possible (Figure 5.9). In total there are 25 states. They are, sequentially: one idle state, two states to instruct the FrameBuffer to read the first byte of the destination address, 6 to read the destination MAC address, 6 to read the source MAC address, 2 to read the Ethernet Type, one to wait for the first payload byte to be written, one for the payload, one to write the last value of the payload, 4 for the CRC32 and the last one to write the Flag 7E as ending of the frame. Also, in the outbound part, the reading and writing of the two different frames are done in parallel. In this case, the crucial moment is the choice of the Protocol field: the second byte of the Ethernet Type field must be read to decide if it is an ARP or IP message. The reading will be done while the second Byte of the Ethernet Type field is written, which means that there are 4 bytes to be read before that. The PPP frame will be written starting from the same state in which the fourth byte of the source address is read. Instead of writing the bytes directly into the FrameBuffer, in this part, they are written into the Shift Memory and then its output is written into the FrameBuffer. Since the code is written using one process and the write enable signal must be asserted when the input data are valid, an extra state before the one used to write the payload must be set. In this way, the ShiftMemory output and the control signal of the FIFO are all synchronized, and all the bytes are transcribed correctly. The output is written directly into the FIFO only when storing the CRC32 bytes, otherwise, it is at first stored into the shiftMemory and its output is written into the FIFO. Once the ReadFrameEnd signal is asserted, the content of the ShiftMemory is shifted to write the last byte of the payload. Then the four bytes of the CRC32 are written and finally the ReadFrame flag is asserted. The FSM goes back to the idle state and the next frame can be processed. When a reading or writing operation is performed, it has to be checked respectively if the FrameBuffer is not empty or the FIFO is not full. In the latter case, there is the need to always check the full signal before taking action: this leads again to have both Moore and Mealy states.[30]



State of FSM	Content read	Content of ShiftMemory	ShiftMemory output	Byte wrote in the FIFO
IDLE	0	7E 7E 7E 7E	7E	X
UPDATE_BUFFER_CONTROL	0	7E 7E 7E 7E	7E	X
WAIT_BUFFER	0	7E 7E 7E 7E	7E	X
RD_DESTINATION_1	destination(47 downto 40)	7E 7E 7E 7E	7E	X
RD_DESTINATION_2	destination(39 downto 32)	7E 7E 7E 7E	7E	X
RD_DESTINATION_3	destination(31 downto 24)	7E 7E 7E 7E	7E	X
RD_DESTINATION_4	destination(23 downto16)	7E 7E 7E 7E	7E	X
RD_DESTINATION_5	destination(15 downto 8)	7E 7E 7E 7E	7E	X
RD_DESTINATION_6	destination(7 downto 0)	7E 7E 7E 7E	7E	X
RD_SOURCE_1	source(47 downto 40)	7E 7E 7E 7E	7E	X
RD_SOURCE_2	source(39 downto 32)	7E 7E 7E 7E	7E	X
RD_SOURCE_3	source(31 downto 24)	7E 7E 7E 7E	7E	X
RD_SOURCE_4_WR_FLAG	source(23 downto 16)	7E 7E 7E 7E	7E	X
RD_SOURCE_5_WR_ADDRESS	source(15 downto 8)	FF 7E 7E 7E	7E	X
RD_SOURCE_6_WR_CONTROL	source(7 downto 0)	03 FF 7E 7E	7E	X
RD_ETHERNET_TYPE_1_WR_PROTOCOL_1	08	00 03 FF 7E	7E	X
RD_ETHERNET_TYPE_2_WR_PROTOCOL_2	00	21 00 03 FF	7E	X
RD_WAIT_WR_PROTOCOL	P1	P1 21 00 03	FF	7E
RD_WR_PAYLOAD	P2	P2 P1 21 00	03	FF
...	...	...	...	...
RD_WR_PAYLOAD	X	C'4 C'3 C'2 C'1	PZ	PY
SHIFT_LAST_OUT	X	X C'4 C'3 C'2	C'1	PZ
WR_CRC_1	X	7E 7E 7E 7E	7E	C1
WR_CRC_2	X	7E 7E 7E 7E	7E	C2
WR_CRC_3	X	7E 7E 7E 7E	7E	C3
WR_CRC_4	X	7E 7E 7E 7E	7E	C4
WR_FLAG	X	7E 7E 7E 7E	7E	7E

Figure 5.9: Data Flow management in the Outbound part of the PPPConverter component. The read content, the content of the ShiftMemory, its output and the byte written in output are shown for each state of the FSM.

5.5.3 Control Flow chart

As well as the inbound part, the Control Flow chart can be found in Appendix Section 14.1. It contains the evolution of the states and all the control signals required. They follow the behavior described in the Data Flow management Section. As for the inbound part, the flow that was followed is shown in Figure 5.4. The Control Flow chart was divided into four parts: the reading of the destination MAC address, the reading of the source MAC address and the writing of the header of the PPP frame, the payload reading and writing, and the CRC and Flag writing.

5.5.4 Timing diagram

The timing diagrams have been separated into four different parts using the same approach as the Control Flow Diagrams also in the outbound part. The four stages that are shown are the start of the reading phase of the Ethernet Frame, the start of the writing phase of the PPP frame, the payload reading and writing phase and the final states of the FSM where the system goes back to the IDLE state. The same rules described for the inbound part apply for the outbound part in terms of content and organization of the diagrams. The trigger for the start of the outbound FSM is when at least a frame is stored inside the FrameBuffer. Then the PPPConverter starts to read it (Figure 5.10). When the proper state is reached, the FSM start to write into the ShiftMemory following the timing interface previously described (Figure 5.11). Then, the FSM requires one state to wait for the payload to be written into the FIFO. When the output of the ShiftMemory is ready and valid it is written into the FIFO according to the previously described timing behavior (Figure 5.12). Finally the system writes directly into the FIFO the CRC bytes, the Flag and goes back to the IDLE state (Figure 5.13).

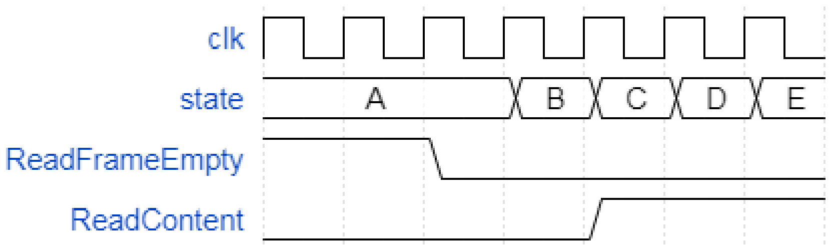


Figure 5.10: Timing diagram of the start of the outbound FSM. In Table 5.5 the coding of the states is shown.

Table 5.5: Coding of the states of the timing diagram in Figure 5.10

State name	Coded name
STATE_IDLE	A
STATE_UPDATE_BUFFER_CONTROL	B
STATE_WAIT_BUFFER	C
STATE_RD_DESTINATION_1	D
STATE_RD_DESTINATION_2	E

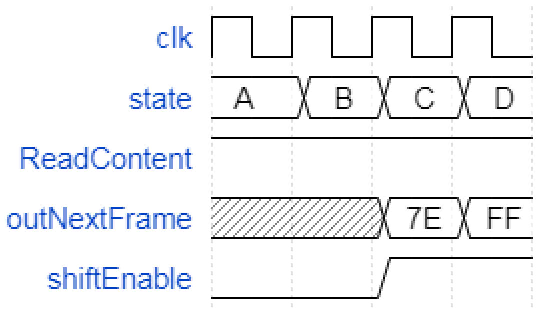


Figure 5.11: Timing diagram of the beginning of the writing of the PPP frame into the FIFO by the outbound part of the PPPConverter component. The coding of the states is shown in Table 5.6.

Table 5.6: Coding of the states for the timing diagram in Figure 5.11

State name	Coded name
STATE_RD_SOURCE_3	A
STATE_RD_SOURCE_4_WR_FLAG	B
STATE_RD_SOURCE_5_WR_ADDRESS	C
STATE_RD_SOURCE_6_WR_CONTROL	D

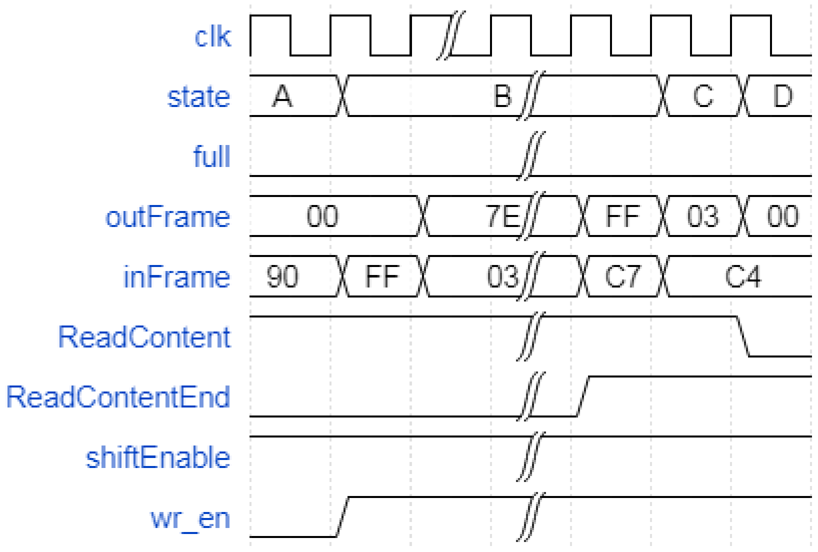


Figure 5.12: Timing diagram of the payload management in the outbound part of the PPPConverter component. The coding of the states is shown in Table 5.7.

Table 5.7: Coding of the state for the timing diagram in Figure 5.12

State name	Coded name
STATE_WAIT_WR_PAYLOAD	A
STATE_RD_WR_PAYLOAD	B
STATE_SHIFT_LAST_OUT	C
STATE_WR_CRC_1	D

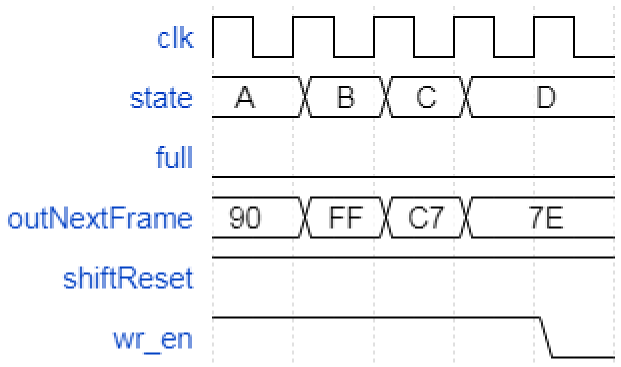


Figure 5.13: Timing diagram of the CRC writing and end frame of the outbound part in the PPPConverter component. The coding of the states is shown in Table 5.8.

Table 5.8: Coding of the state for the timing diagram in Figure 4.13

State name	Coded name
STATE_WR_CRC_3	A
STATE_WR_CRC_4	B
STATE_WR_FLAG	C
STATE_IDLE	D

6. Simulations of PPPConverter

To analyze the behavior of the designed PPP, this component has been simulated with the Xilinx tool ISE 14.7. Since there is a Core Generated component, all the simulations have been done in the ISE 14.7 tool, Xilinx property. All the testbench codes used in this section can be found in the Appendix Section 14.4 and the PPPConverter code can be found in Appendix Section 14.3. For sake of brevity, here only the waveform results will be shown. The bytes used as MAC addresses, payload and CRC are all random numbers.

6.1 ShiftMemory simulations

It is possible to demonstrate that the ShiftMemory component behaves as expected when stimulated with a proper testbench (Figures 6.1 and 6.2). The testbench used to simulate the behavior of the ShiftMemory component is based on a process with a *wait for* statement. At each clock cycle, one more byte is written into the component as the shiftEnable signal is asserted. When the shiftEnable signal is not asserted, the ShiftMemory stops to shift in values. If the shiftReset signal is asserted, the expected content of all the registers became 0x00 if in the inbound part (Figure 6.1) or 0x7E in the outbound part (Figure 6.2). The testbench that can be found in Chapter 14 Section 14.4.1.

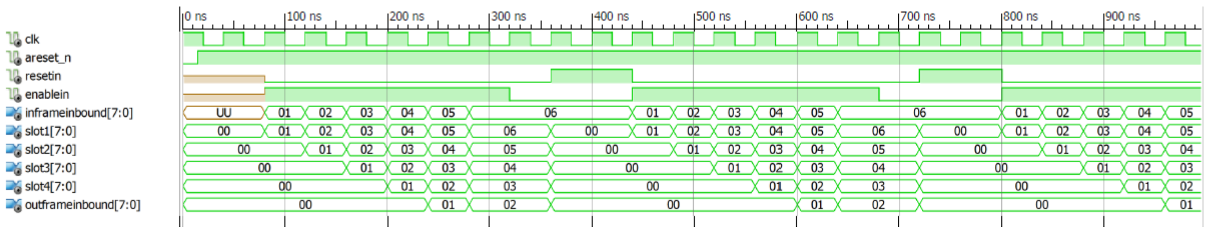


Figure 6.1: Waveform resulting from the testbench applied on the inboundShiftMemory.

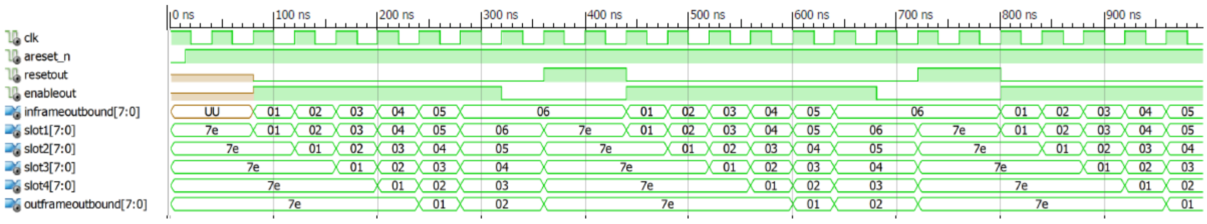


Figure 6.2: Waveform resulting from the testbench applied on the outboundShiftMemory.

6.2 PPPConverter simulations results

To verify that the VHDL code of the PPPConverter satisfies the expected behavior, it was stimulated with a custom testbench. This testbench generates the clock signal, the asynchronous reset and the stimuli sequences for the inbound and outbound part. Regarding these two parts, two specific processes were used so that it was possible to test the full duplex behavior of the component.

6.2.1 Inbound part simulation results

The inbound part of the PPPConverter component was simulated with a testbench and the resulting behavior is as expected. The testbench is based on wait for statements and it recreates the interface of the inboundFIFO. When the empty signal is not asserted, the inbound part starts to write the destination (Figure 6.3) and source MAC addresses as well as reading from the dummy FIFO the PPP frame (Figure 6.4). Then the payload is written as it is in the FrameBuffer (Figure 6.5) and finally when the 0x7E byte is read, the FSM writes the new CRC signature and return to the IDLE state (Figure 6.6). The testbench code is in Appendix Section 14.4.2.

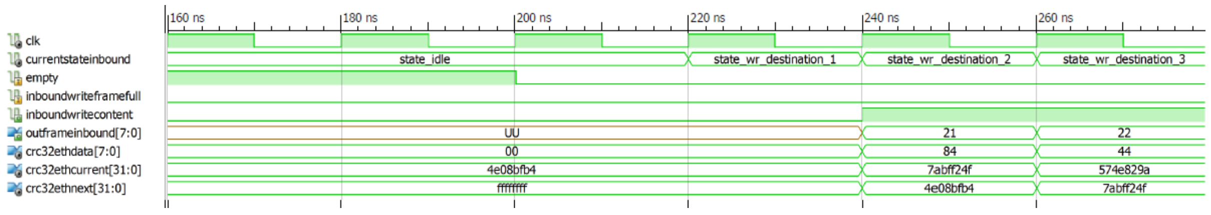


Figure 6.3: Simulation results of the inbound part. Start of the FSM and writing of the destination MAC address.

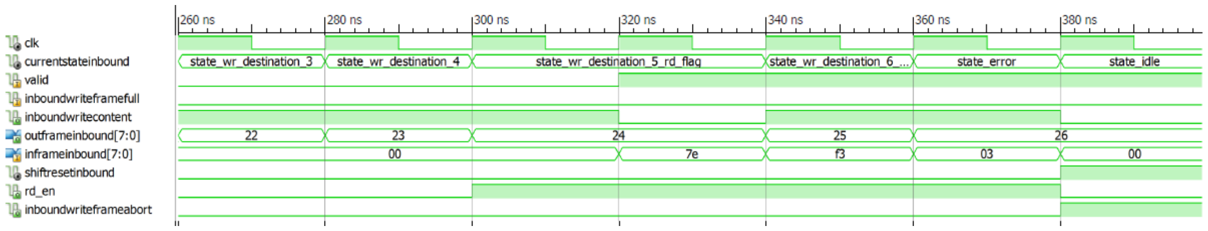


Figure 6.4: Simulation results of the inbound part. The PPP frame is read, and a wrong value is received. The FSM aborts it and goes back to the IDLE state.

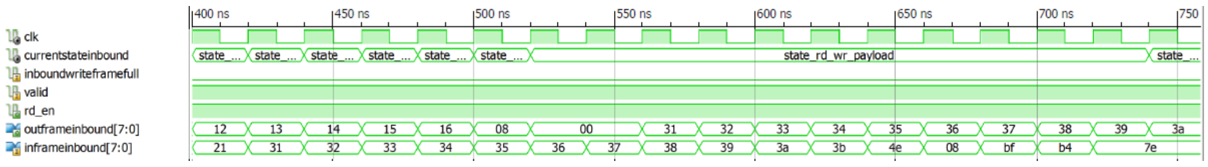


Figure 6.5: Simulation results of the inbound part. The payload is read and written. When the byte 7E is received the FSM starts to write the CRC.

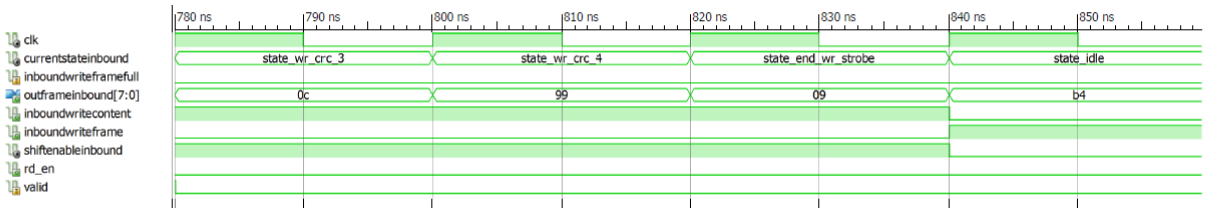


Figure 5.6: Simulation results of the inbound part. The last two bytes of the CRC are written and the FSM goes back to the IDLE state ready to convert another frame.

6.2.2 Outbound part simulation results

The testbench used to stimulate the outbound part was based as well on *wait for* statements process and it recreates the FrameBuffer interface. When the FrameBuffer is not empty, the outbound part starts to read the destination MAC address (Figure 6.7). While reading the source address it starts to write the PPP frame into the outboundShiftMemory (Figure 6.8). The payload, then, is transferred as it is until the ReadFrameEnd signal is asserted (Figure 6.9). Finally, the new CRC signature is written into the outboundFIFO as well as the flag and the FSM goes back to the IDLE state (Figure 6.10). The testbench code can be found as well in Appendix Section 14.4.2.

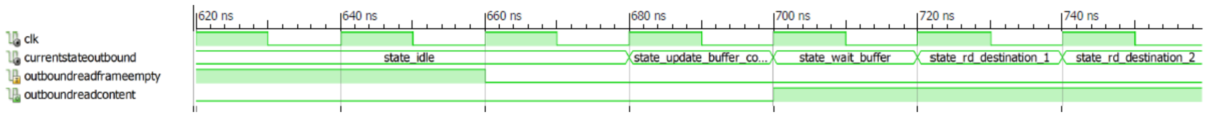


Figure 6.7: Simulation results of the outbound part. The FSM starts to read the destination MAC address.

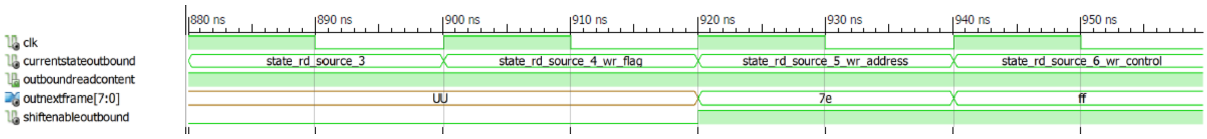


Figure 6.8: Simulation results of the outbound part. The PPP frame is written.

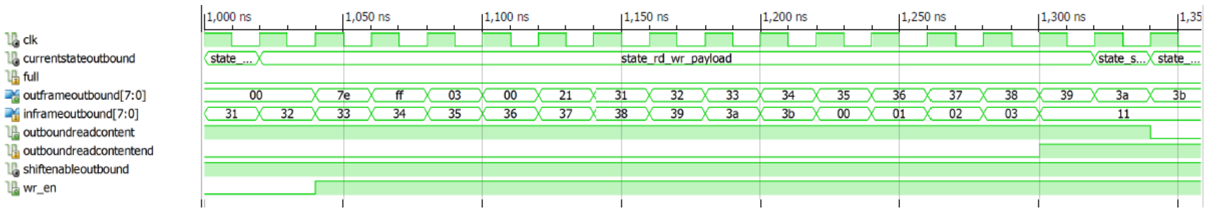


Figure 6.9: Simulation result of the outbound part. The Payload is read and written until the proper signal from the FrameBuffer is asserted.

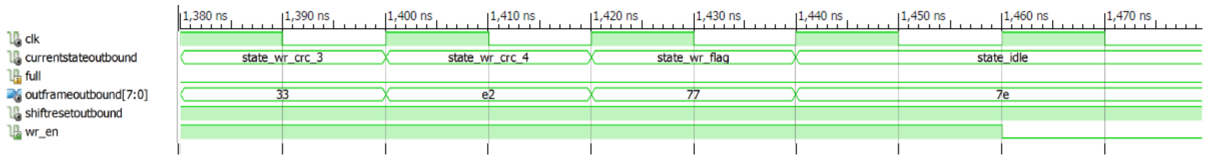


Figure 6.10: Simulation result of the outbound part. The CRC bytes are written and the FSM goes back to the IDLE state.

7. Implementation and test on physical board

The implementation of the PPPConverter featured multiple steps. Firstly, a simulation of the full chain of the message through two boards was performed with successful results. Secondly, only the PPPConverter chain was uploaded on the FPGA to check if the system was transmitting the message through the IOChannel. Then the PPPConverter chain was uploaded in parallel to the existent system to check if it could still work properly when the IOChannel is transmitting data. Finally, the whole new system was uploaded and tested.

7.1 The simulation of the implemented system and results

The simulation of a full chain implementation was performed. The stimuliProcess a data-generator process that simulates a random ARP request from the component belonging to a BoardA to the ones belonging to a BoardB (Figure 7.1). In this picture is also shown the path of the message and all the components it will go through. When the message is generated in the BoardA, it is written into the Rgmii-PHY component, sent to the Bridge and written into the FrameBuffer connected to the Port 0 of the Ethernet Switch. Since the message is broadcast by definition, it is sent to all the ports including the one with the PPPConverter component. The message is written into the FrameBuffer to which the PPPConverter reads and translate the Ethernet Frame to a PPP frame. The frame is then written into the UARTInterface component which serialized it. The serial signal is the input of the UARTInterface belonging to the BoardB. The received bytes are stored in the FIFO and read by the PPPConverter which recreates the Ethernet Frame and save it into the FrameBuffer. This buffer is connected to another Ethernet Switch which sends the broadcast message to all the ports. The FrameBuffer of the Port 1 is written and, when a full frame is stored, the Bridge reads it, attach the Preamble and SFD fields to it and sends it to the Rgmii-PHY component. The latter should then send the message to the processor.

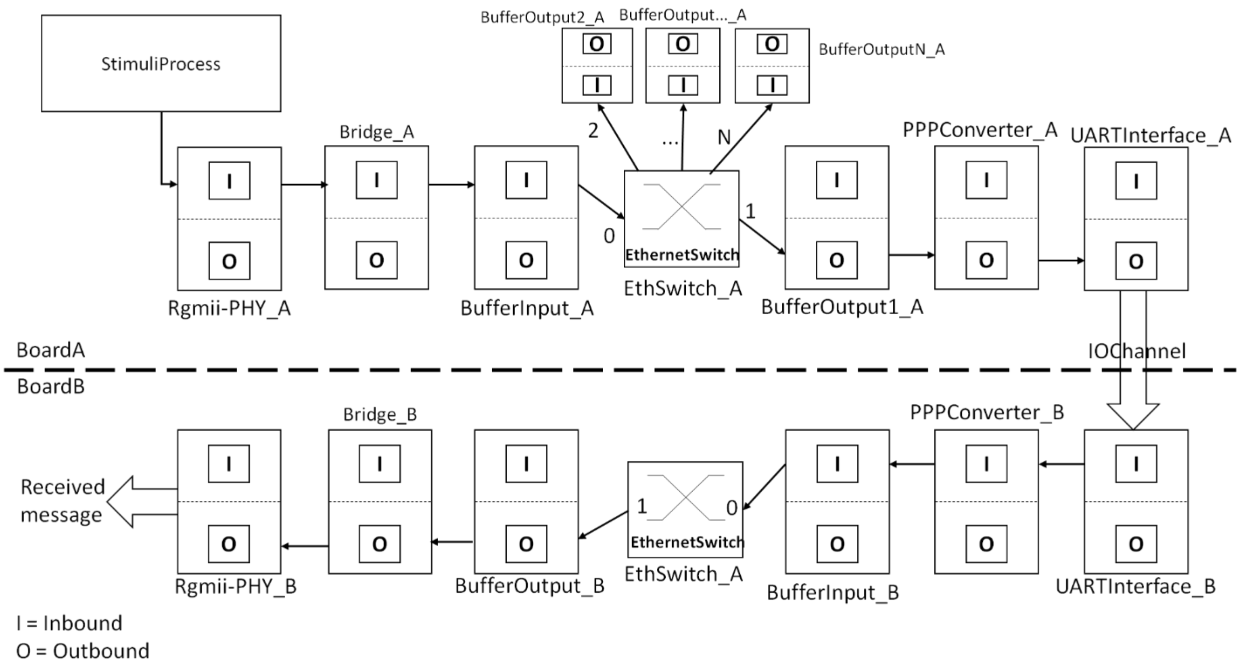


Figure 7.1: The DataPath of the full chain of transmission. This scheme was followed while developing the testbench to simulate the behavior of a physical board.

With the stimuliProcess shown in Appendix 14.4.3, the full chain behavior was simulated with successful results. The testbench recreates a Rgmii frame and writes it into the Rgmii-PHY component. The testbench is developed with an FSM in which at each state a four-bit value is sent. In these simulations, the bytes that belong to the message are purely random. In all this Figure and the two consecutive, the Rgmii-PHY component is called SPhy, where S stands for Server, as the master of the communication.

The result of the simulation prove that the full chain transmission works as expected. As the transmission starts, the four-bites data are sent together with the tx_ctl signal asserted. The data written into the BufferInput_A (Figure 7.2). The message is then received by the PPPConverter, it is translated and the IOChannel start to send the message in a serial way (Figure 7.3). Finally, the frame is received on the outboundData_i signal of the Rgmii-PHY as the final stage of the whole chain transmission (Figure 7.4). In Figure 7.5 is shown the full chain transition where the message is sent from the Board A and received on the Board B.

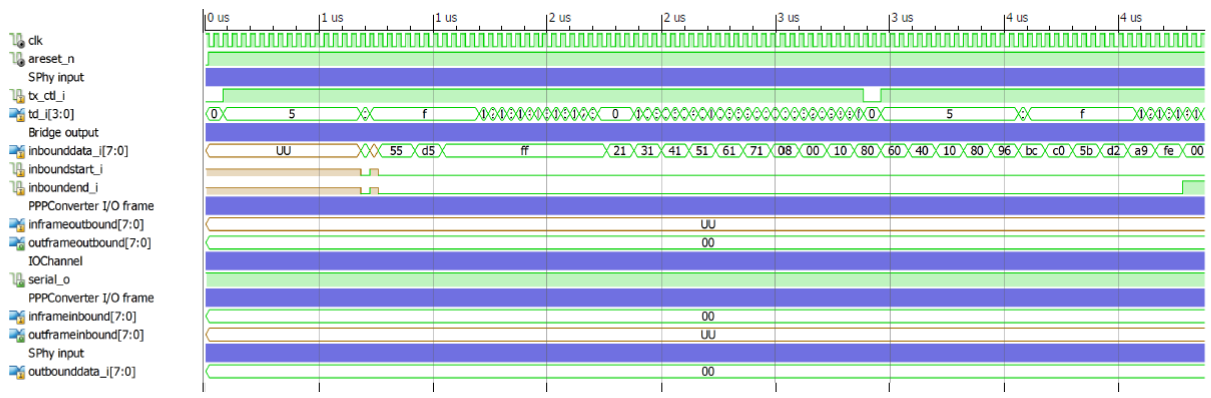


Figure 7.2: Start of the transmission of the frame. The message is received by the Rgmii-Phy component, passed through the Bridge and written into the BufferInput_A. The violets lines are the divisors and they indicate the meaning of the signals right below them.

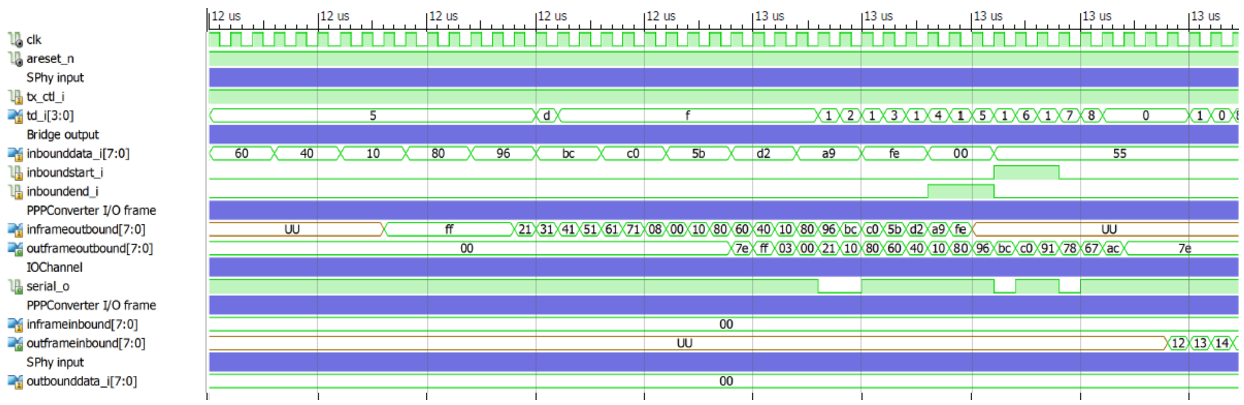


Figure 7.3: Simulation results of the full chain. The PPPConverter on the Board A receives the frame and translate it. The PPP frame is then serialized by the UARTInterface in the Board A. It can be also noticed the start of the conversion of the message by the PPPConverter on the Board B. The violets lines are the divisors and they indicate the meaning of the signals right below them.

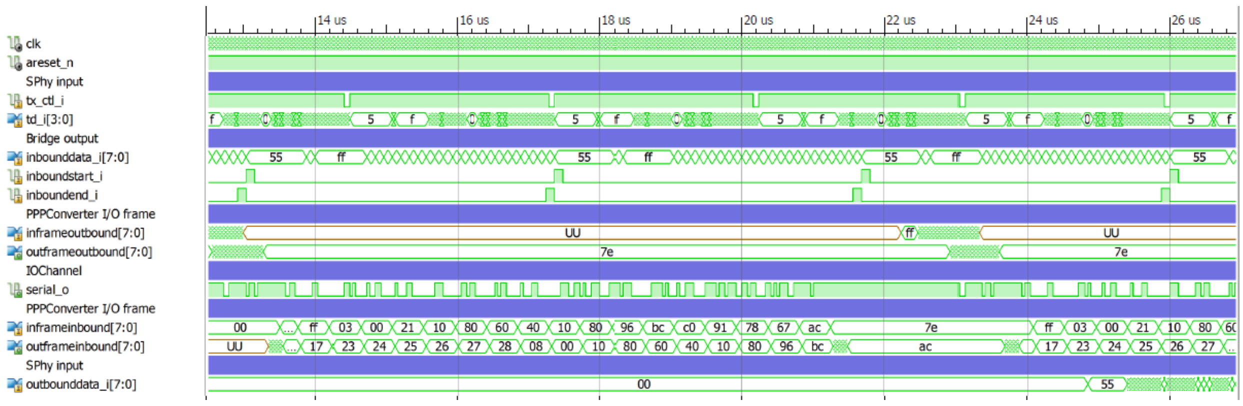


Figure 7.4: Simulation results of the full chain. The message is sent through the IOChannel and it is sent along the chain until being written into the Rgmii-PHY. It is possible to notice the conversion from PPP frame to Ethernet in the PPPConverter component. The violets lines are the divisors and they indicate the meaning of the signals right below them.

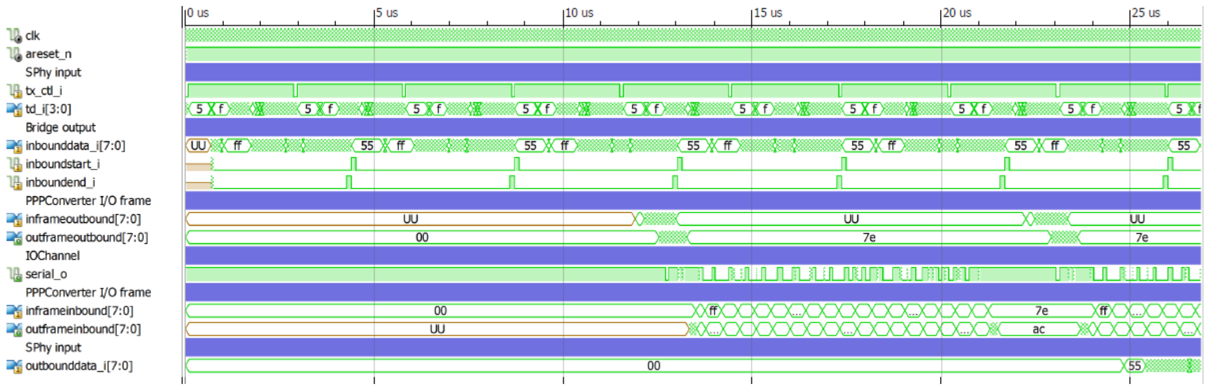


Figure 7.5: Simulation results of the full chain. The frame is sent from the Board A and it reaches finally the Rgmii-PHY component on the Board B. The violets lines are the divisors and they indicate the meaning of the signals right below them.

7.2 The implementation of the PPPConverter chain and results

The FPGA was programmed to have only the PPPConverter component as well as a FrameBuffer, and the UARTInterface to check if a waveform was transmitted on the physical IOChannel (Figure 7.6). The test produced the desired waveform as output of the IOChannel. To obtain the data, a process called stimuliProcess was used as a testbench to write data into the FrameBuffer (Appendix 14.4.3). The result of this physical test is successful, a waveform was captured on the IOChannel. Since the link is LVDS, the reference probe was inserted on the *negate* link and the signal probe on the *positive* link. The sampling of the data was made with athe Mixed Signal Oscilloscope MSO 70404C by Tektronix (Figure 7.7). The clock frequency chosen for this test is random and does not have meaning in the actual performances of the IOChannel.

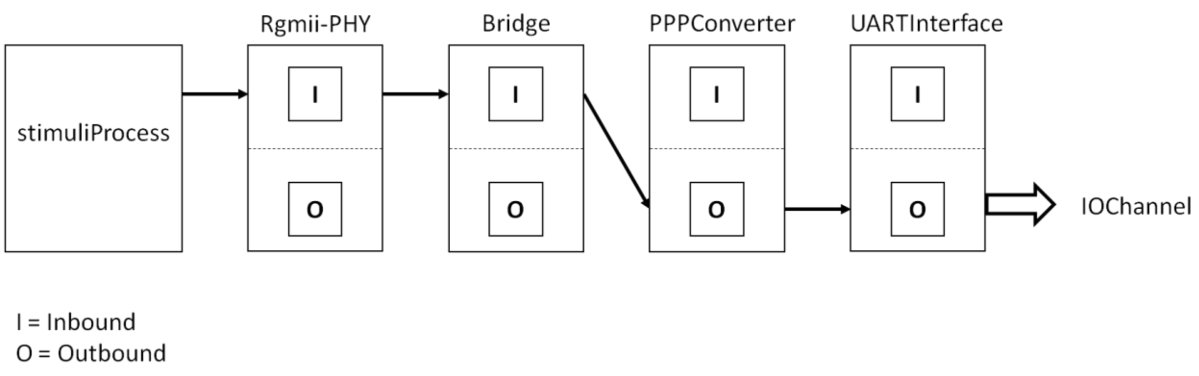
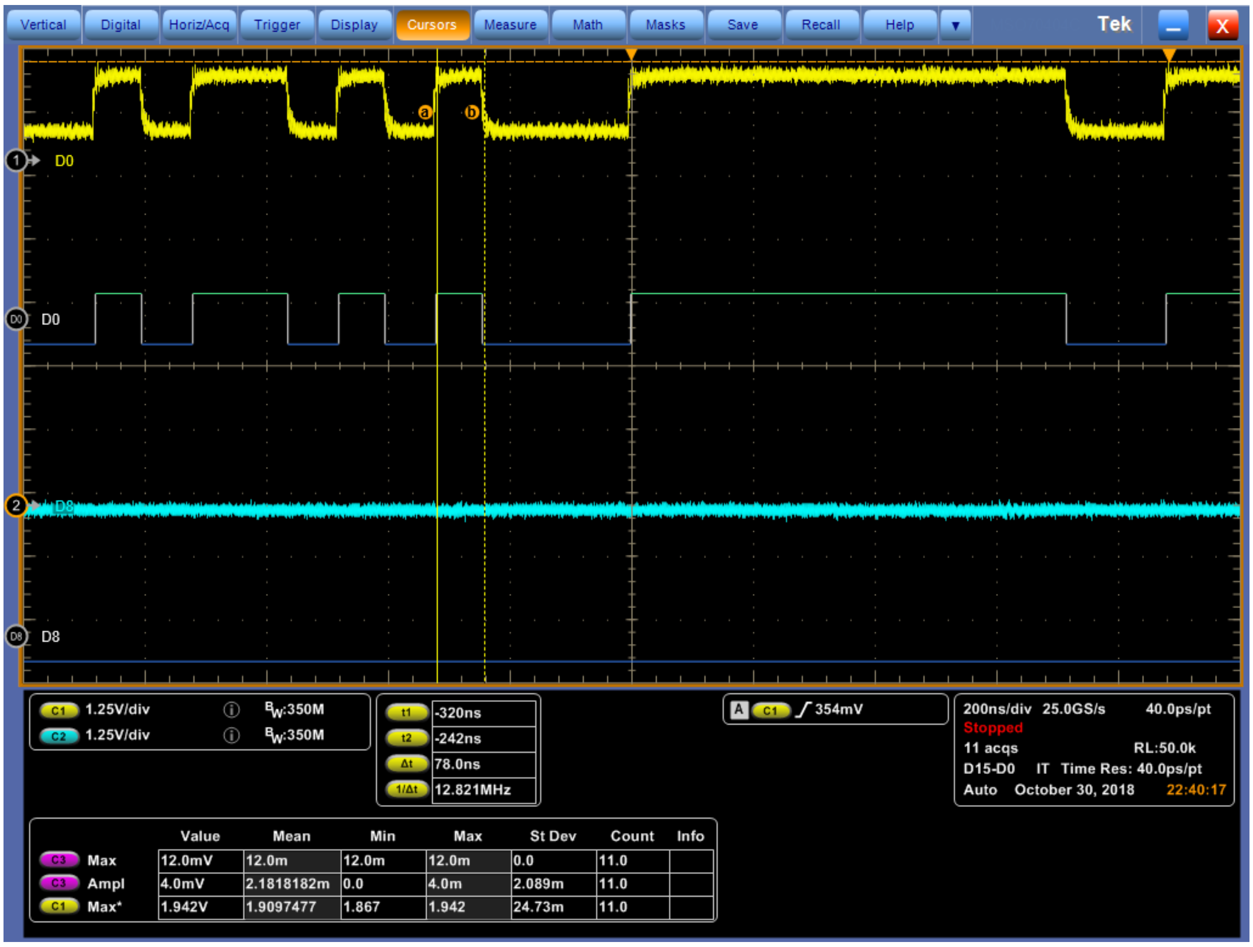


Figure 7.6: First test implemented on the board, the chain involves only the PPPCoverter chain blocks



```
C1 1.25V/div   BW:350M
C2 1.25V/div   BW:350M

t1    -320ns
t2    -242ns
Δt    78.0ns
1/Δt  12.821MHz

A C1 ∫ 354mV

200ns/div  25.0GS/s      40.0ps/pt
Stopped
11 acqs                  RL:50.0k
D15-D0    IT  Time Res: 40.0ps/pt
Auto   October 30, 2018    22:40:17
```

	Value	Mean	Min	Max	St Dev	Count	Info
C3 Max	12.0mV	12.0m	12.0m	12.0m	0.0	11.0	
C3 Ampl	4.0mV	2.1818182m	0.0	4.0m	2.089m	11.0	
C1 Max*	1.942V	1.9097477	1.867	1.942	24.73m	11.0	

Figure 7.7: Logic Analyzer sampled waveform representing the output of the IOChannel produced with the setup in Figure 7.2. The frequency value is random

7.3 The implementation of the parallel design and results

The existing system and the PPPConverter chain were uploaded on the same FPGA but not linked. The PPPConverter is still stimulated by the stimuliProcess. To check if the existing system could work in this setup, the PING utility was used. The two physical boards are connected as in the already existent system and the PING request was sent to the other board. The same structure was uploaded on the two boards. The results of this test were not successful: the PING utility could not send any message to the other board. This issue was checked with the tcpdump instruction[29], which shows on the terminal what messages are sent or received by the processors. The IOChannel, though, was still transmitting the stream of data as expected (Figure 7.9).

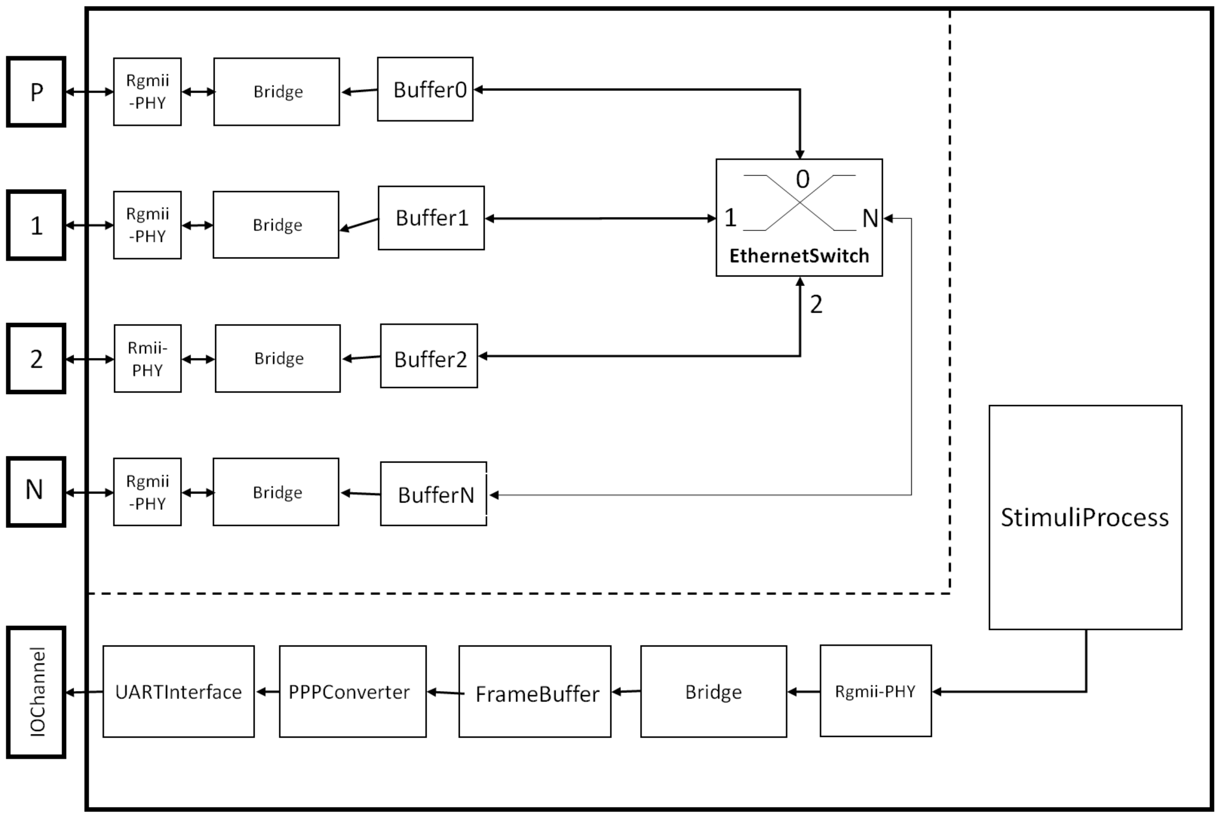
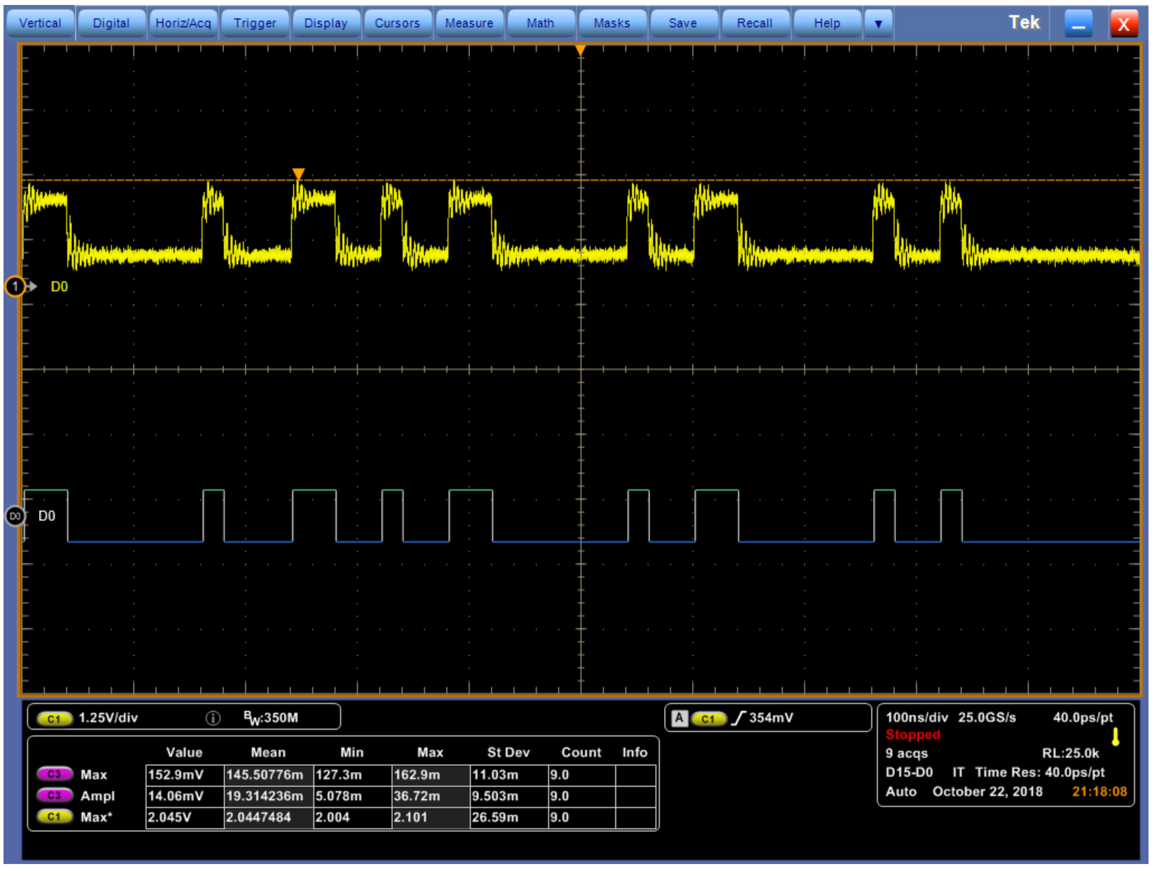


Figure 7.8: Parallel implementation test setup. The system inside the dashed line is the system already implemented while the other is the PPPConverter chain. The component called P is the Processor.



C1 1.25V/div BW:350M

A C1 354mV

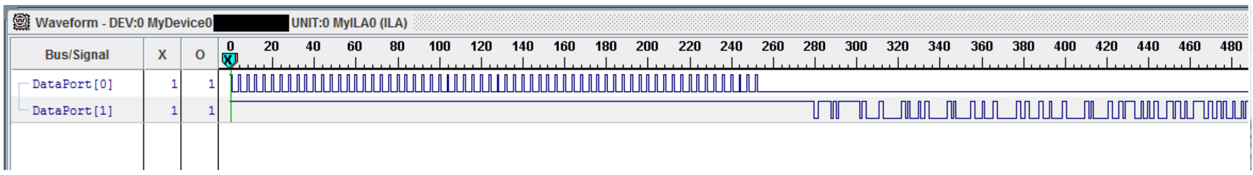
		Value	Mean	Min	Max	St Dev	Count	Info
C3	Max	152.9mV	145.50776m	127.3m	162.9m	11.03m	9.0	
C3	Ampl	14.06mV	19.314236m	5.078m	36.72m	9.503m	9.0	
C1	Max*	2.045V	2.0447484	2.004	2.101	26.59m	9.0	

```
100ns/div  25.0GS/s        40.0ps/pt
Stopped
9 acqs                      RL:25.0k
D15-D0    IT  Time Res: 40.0ps/pt
Auto   October 22, 2018     21:18:08
```

Figure 7.9: Logic Analyzer sampled waveform representing the output of the IOChannel produced with the setup in Figure 7.8. The frequency of the signal is random

7.4 Results of the integration on the physical board

The PPPConverter chain was integrated into the existing architecture by connecting the FrameBuffer to one of the Ports of the EthernetSwitch. The code was compiled, and the resulting bit file was uploaded on the two Boards. The PING utility was used again to test if the two boards could communicate through the IOChannel. The test was unsuccessful since it showed that the PING request was not sent to the other board. The ChipScope PRO tool was used to check the behavior of the system inside the FPGA. This tool is designed by Xilinx and allows to have the sampled waveform of some signals inside the FPGA. The signals are shown together with a trigger signal that starts the sampling of the data. For this test, the asserted wr_en signal of the outboundFIFO was the trigger and the data was the IOChannel serial signal sent to the buffer that translates the message from single-ended to differential. This differential signal is then sent toward the LVDS pins of the FPGA. The analysis with ChipScope showed that on the input of the differential buffer there was an IOChannel signal (Figure 7.10), but the board could not propagate it toward the physical IOChannel since the Mixed Signal Oscilloscope could not sample any data.



Waveform - DEV:0 MyDevice0 [redacted] UNIT:0 MyILA0 (ILA)

Bus/Signal	X	O
DataPort[0]	1	1
DataPort[1]	1	1

Figure 7.10: Chiscope results. The DataPort (0) is the trigger signal wr_en of the outboundFIFO and the DataPort (1) is the IOChannel serial signal

8. Discussion and future implementations

The results of the simulation of the different part of the PPPConverter prove that it was developed according to the expected behavior. The ShiftMemory component shows the diagonal shift behavior as the timing diagram was expecting it to behave. The inbound part correctly verifies that the incoming PPP frame has the correct format and translate it into an Ethernet frame. It also interfaces correctly the FrameBuffer and FIFO since no byte is lost or duplicated in the communication. The outbound part also transforms the Ethernet frame into a PPP one without modifying the payload. However, there are still three important features that have to be implemented in future stages in the VHDL code. These allow the component to be flexible and fully compliant of the PPP standard. The MAC addresses are hardcoded in the VHDL as constant signals, which is not a flexible solution if the board is substituted. The escaping sequence belonging to the PPP standard has not been developed to simplify the initial development of the PPPConverter but it must be developed to create a fully standard interface. Finally, the FIFO used in the UARTInterface is a Xilinx Core generated FIFO. This means that if a new FPGA is chosen with a different brand, the whole interface has to be redesigned.

The MAC addresses are different depending on which part of the PPPConverter is considered. In the inbound part, the source MAC address is a locally administered one is can be hard-coded inside the system since it is used only locally. However, the destination MAC address is the universally administered of the processor. This value cannot be hardcoded since this will result in a custom made code for each sold board. To prevent this to happen, in future implementation the destination address must be saved in a register that could communicate also with the outbound part. When the message is sent outside the board, the source address is the one identifying the processor. Therefore, when an outbound frame is detected, the source field should be saved so that the inbound part can update the correct address. This operation should be developed outside of the PPPConverter component in order to keep the component full duplex.

The escaping mechanism should be added to the FSMs of both inbound and outbound parts. In the outbound part, whenever a byte is read, if it is equal to 0x7E it should be substituted with a sequence of two bytes: 0x7D and the XOR between 0x7E and 0x20. If the byte 0x7D is read, it is substituted with 0x7D and the XOR between x7D and 0x20. In the inbound part, if a 0x7D byte is received, it means that one byte was escaped. The resulting byte will be the either the XOR between 0x7D and 0x20 or between 0x7E and 0x20. This verification will be added whenever a new byte is read. It will also require to modify the FSM in both parts by adding one state to handle the extra inserted byte. This solution will not compromise the synchronization toward either the FrameBuffer or the FIFOs. This escaping sequence will allow the system to be fully standard and it will prevent the interruption of the frame.

The Core generated FIFO should be removed and substituted with a generic FIFO. The used FIFO is generated by the Xilinx ISE 14.7 tool and it can only be tested in that environment. Moreover, if future implementations of the system will require an FPGA sold by a different brand, the whole interface must be redesigned in terms of timing and signal interface. It might even be that the PPPConverter should be designed again from the start if there is a significant difference. Developing a brand-independent FIFO will ease the design in term of the flexibility of the system.

Concerning the physical implementation of the interface, the results show that the system does not work. The single PPPConverter chain is reacting as expected since waveforms were sampled on the IOChannel, but the integration with the existent system must be thoroughly studied. The problem might be generated by some timing constraints not related to the internal logic. Since there was no warning from the ISE tool about setup and hold times, the synchronization with other elements on the board that are connected to the FPGA must be analyzed. If more logic is added to the FPGA the system is going to react more slowly even if the new components are not connected to the previous system. Moreover, since the IOChannel was not used before, it can also be that the processor sets some part of the board in a status that forces the IOChannel to stop to communicate when connected to the message flow. This hypothesis is made by analyzing the ChipScope image and noticing that the Logic Analyzer could not observe any signal. The FPGA is producing a serial stream of data but it does not reach the physical pin.

9. Conclusions

The PPPConverter component was developed in VHDL. The relationship between it and the components already in the system was studied and taken into consideration during the development of the VHDL code. The timing and signal requirements were satisfied. The two parts were developed to reach the most parallelized possible flow of data as well as the control of the integrity of the signal. A new component, the ShiftMemory, was developed to further synchronize the conversion of the frame. The simulations of the PPPConverter and the ShiftMemory are successful since they behave with the theoretic expected results. The physical implementation is not fully complete.

With this new component, the system will not only be able to communicate via a new channel that is faster than the one used before but also maintains the integrity of the messages. However, there are some limitations to the component and the system. The PPPConveter is not fully developed as it lacks a proper MAC address management. Moreover, the PPP standard is not fully implemented yet since the escaping function is not developed in the code. This will add more integrity to the transmission of the messages. There are also some limitations on the system level. The FIFO is specifically used for the used FPGA and it should be developed as a generic component to increase the flexibility of the system. Moreover, the physical implementation shows timing constraint issues which should be studied thoroughly as well as the processor capability of control of the board elements.

Overall, since this component is the first interface developed for the IOChannel, it is a good starting point. Simulations and its own physical implementation satisfy the requirement. Although, since the system in which it is inserted is complex, deeper studies and development must be carried out.

10. List of Symbols

Name	Meaning
ARP	Address Resolution Protocol
CBI	Computer-based Interlocking System
CIS	Central Interlocking System
CRC	Cyclic Redundant Check
FIFO	First In First Out
FPGA	Field Programmable Gate Array
HDLC	High-level Data Link Control
IEEE	Institute of Electrical and Electronics Engineers
IP	Internet Protocol
Ipv4	Internet Protocol version 4
LAN	Local Area Network
LSB	Least Significant Bit
LLC	Logical Link Control
LVDS	Low Voltage Differential Signal
MAC	Media Access Control
MAN	Metropolitan Area Network
OCS	Object Controller System
OSI	Open System Interconnection
PCB	Printed Circuit Board
PDU	Protocol Data Unit
PHY	PHYsical Layer
PING	Packet Internet Grouper
PPP	Point to Point Protocol
RAMS	Reliability, Availability, Maintainability, Safety
SFD	Start of Frame Delimiter
SIL	Safety Integrity Level
SLIP	Serial Line Interface Protocol
TCC	Traffic Control Center
TMS	Traffic Management System
TN	Transmission Network
UART	Universal Asynchronous Receiver-Transmitter
VHDL	Very High Speed Integrated Circuits Hardware Description Language
WAN	Wide Area Network

11. List of Figures

12. List of Tables

13. References

[1] "The Components of a Telecommunications System - Video &amp; Lesson Transcript | Study.com." [Online]. Available: https://study.com/academy/lesson/the-components-of-a-telecommunications-system.html. [Accessed: 28-Nov-2018].

[2] A. Zaklouta, "High-Speed Communication in OSI Layer 2 Research and Implementation," KTH, 2018.

[3] "What is an FPGA? Field Programmable Gate Array." [Online]. Available: https://www.xilinx.com/products/silicon-devices/fpga/what-is-an-fpga.html. [Accessed: 02-Dec-2018].

[4] "What Is a Computer Network Switch?" [Online]. Available: https://www.lifewire.com/definition-of-network-switch-817588. [Accessed: 29-Nov-2018].

[5] "Understanding LVDS for Digital Test Systems - National Instruments." [Online]. Available: http://www.ni.com/white-paper/4441/en/. [Accessed: 29-Nov-2018].

[6] "IEEE 802.3-2018 - IEEE Approved Draft Standard for Ethernet." [Online]. Available: https://standards.ieee.org/standard/802_3-2018.html. [Accessed: 29-Nov-2018].

[7] W. Simpson, "The Point-to-Point Protocol (PPP)."

[8] J. L. Romkey, "Nonstandard for transmission of IP datagrams over serial lines: SLIP."

[9] "[MS-ABS]: 32-Bit CRC Algorithm." [Online]. Available: https://msdn.microsoft.com/en-us/library/dd905031.aspx. [Accessed: 29-Nov-2018].

[10] Bombardier Inc., "History of Bombardier - Planes and Trains," 2014. [Online]. Available: https://www.bombardier.com/en/about-us/history.html. [Accessed: 29-Nov-2018].

[11] "Frecciarossa 1000 Very High Speed Train Makes Maiden Journey in Italy - Bombardier." [Online]. Available: https://www.bombardier.com/en/media/newsList/details.bombardier-transportation20150427frecciarossa1000veryhighspeedtr.bombardiercom.html. [Accessed: 29-Nov-2018].

[12] "Bombardier's INNOVIA APM 300 Automated People Mover System Enters Service at Munich Airport - Bombardier." [Online]. Available: https://www.bombardier.com/en/media/newsList/details.bt-20160422-bombardiers-innovia-apm-300-automated-people-mover-s.bombardiercom.html. [Accessed: 29-Nov-2018].

[13] "What is the OSI Model? - Definition from Techopedia." [Online]. Available: https://www.techopedia.com/definition/24205/open-systems-interconnection-model-osi-model. [Accessed: 29-Nov-2018].

[14] "What is Data Link Layer? - Definition from Techopedia." [Online]. Available: https://www.techopedia.com/definition/18698/data-link-layer. [Accessed: 29-Nov-2018].

[15] W. Peterson and D. Brown, "Cyclic Codes for Error Detection," *Proc. IRE*, vol. 49, no. 1, pp. 228–235, Jan. 1961.

[16] "File:UART timing diagram.svg - Wikimedia Commons." [Online]. Available: https://commons.wikimedia.org/wiki/File:UART_timing_diagram.svg. [Accessed: 29-Nov-2018].

[17] "Standard Group MAC Addresses Standard Group MAC Addresses: A Tutorial Guide."

[18] "B1R-623 Cisco IOS Bridging and IBM Networking Command Reference, Volume 1 of 2 Ethernet Type Codes."

[19] J. Postel, "Internet Protocol."

[20] "Point-to-Point (PPP) Protocol Field Assignments." [Online]. Available: https://www.iana.org/assignments/ppp-numbers/ppp-numbers.xhtml. [Accessed: 29-Nov-2018].

[21] D. Plummer, "An Ethernet Address Resolution Protocol: Or Converting Network Protocol Addresses to 48.bit Ethernet Address for Transmission on Ethernet Hardware."

[22] "man page ping section 8." [Online]. Available: http://www.manpagez.com/man/8/ping/. [Accessed: 29-

Nov-2018].

[23] "Wayback Machine." [Online]. Available: https://web.archive.org/web/20160303171328/http:/www.hp.com/rnd/pdfs/RGMIIv2_0_final_hp.pdf. [Accessed: 29-Nov-2018].

[24] "What is a network bridge? | CCNA." [Online]. Available: https://geek-university.com/ccna/what-is-a-network-bridge/. [Accessed: 29-Nov-2018].

[25] "Wishbone :: OpenCores." [Online]. Available: https://opencores.org/howto/wishbone. [Accessed: 30-Nov-2018].

[26] "round robin arbitration | RTLery." [Online]. Available: http://www.rtlery.com/articles/round-robin-arbitration. [Accessed: 30-Nov-2018].

[27] "LogiCORE IP FIFO Generator v9.3 Product Guide PG057," 2012.

[28] "Weblet Importer." [Online]. Available: https://www.tcpdump.org/manpages/tcpdump.1.txt. [Accessed: 29-Nov-2018].

[29] "tcpdump(8): dump traffic on network - Linux man page." [Online]. Available: https://linux.die.net/man/8/tcpdump. [Accessed: 29-Nov-2018].

[30] "Moore and Mealy Machines." [Online]. Available: https://www.tutorialspoint.com/automata_theory/moore_and_mealy_machines.htm. [Accessed: 29-Nov-2018].

14. Appendix

14.1 Inbound Control Flow charts

In the following charts are represented the behavior of the inbound FSM divided in the four parts as described in Figure 5.4. There are also the same charts with the explicit signals. In these charts are shown only the signals that change in that state and have an effect on the behavior of the system.

14.1.1 Preamble

In Figures 14.1 and 14.2 are shown respectively the start of the FSM together the writing of the destination MAC address and the start of the reading part of the PPP frame. In Figures 14.3 and 14.4 are represented the same charts but with the explicit controls used in every state.

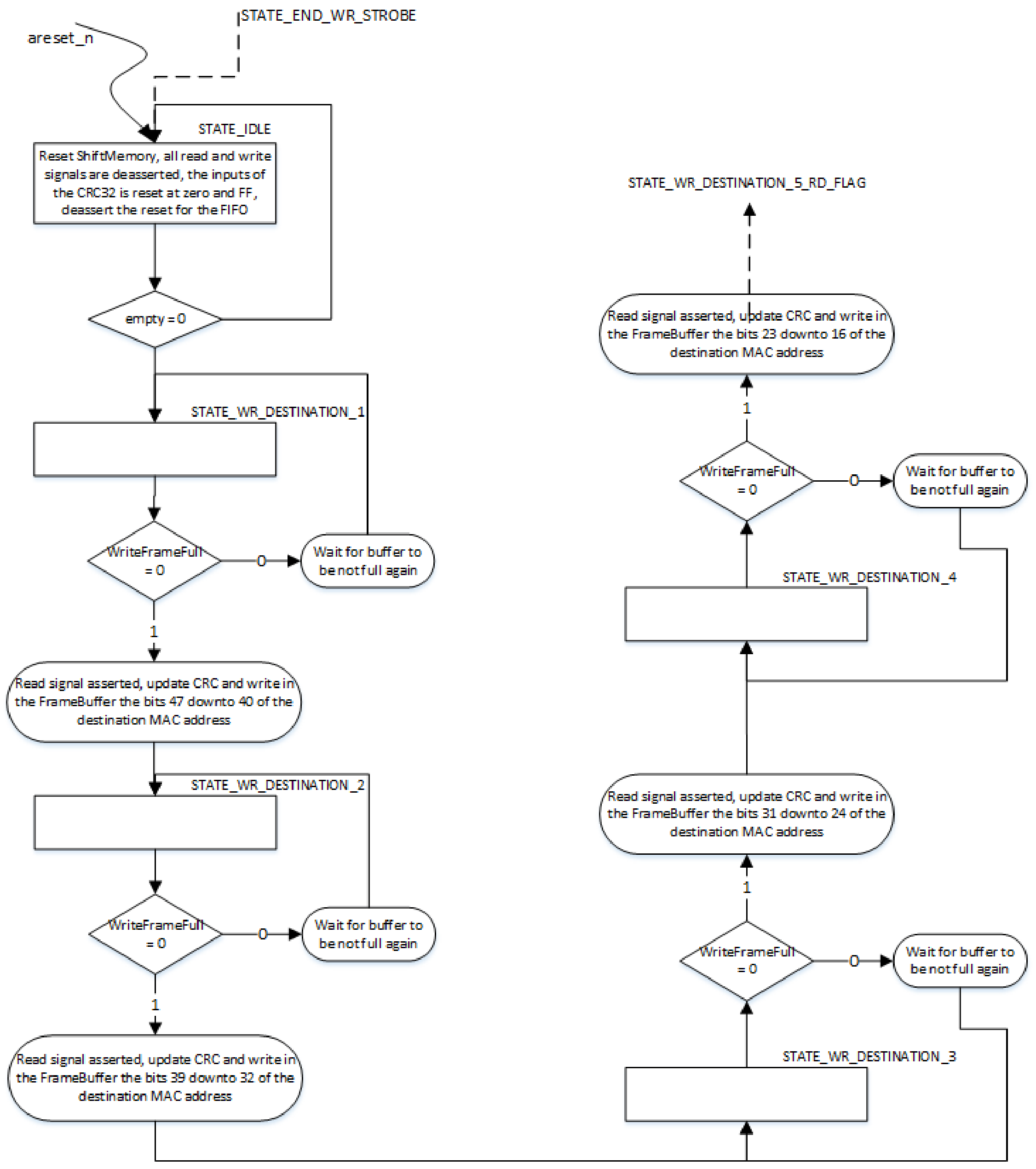

STATE_END_WR_STROBE
areset_n
STATE_IDLE
Reset ShiftMemory, all read and write signals are deasserted, the inputs of the CRC32 is reset at zero and FF, deassert the reset for the FIFO
empty = 0
STATE_WR_DESTINATION_1
WriteFrameFull = 0
0
Wait for buffer to be not full again
1
Read signal asserted, update CRC and write in the FrameBuffer the bits 47 downto 40 of the destination MAC address
STATE_WR_DESTINATION_2
WriteFrameFull = 0
0
Wait for buffer to be not full again
1
Read signal asserted, update CRC and write in the FrameBuffer the bits 39 downto 32 of the destination MAC address
STATE_WR_DESTINATION_5_RD_FLAG
Read signal asserted, update CRC and write in the FrameBuffer the bits 23 downto 16 of the destination MAC address
1
WriteFrameFull = 0
0
Wait for buffer to be not full again
STATE_WR_DESTINATION_4
Read signal asserted, update CRC and write in the FrameBuffer the bits 31 downto 24 of the destination MAC address
1
WriteFrameFull = 0
0
Wait for buffer to be not full again
STATE_WR_DESTINATION_3

Figure 14.1: Control flow chart, behavioral representation. Part 1 of the Preamble in the inbound part

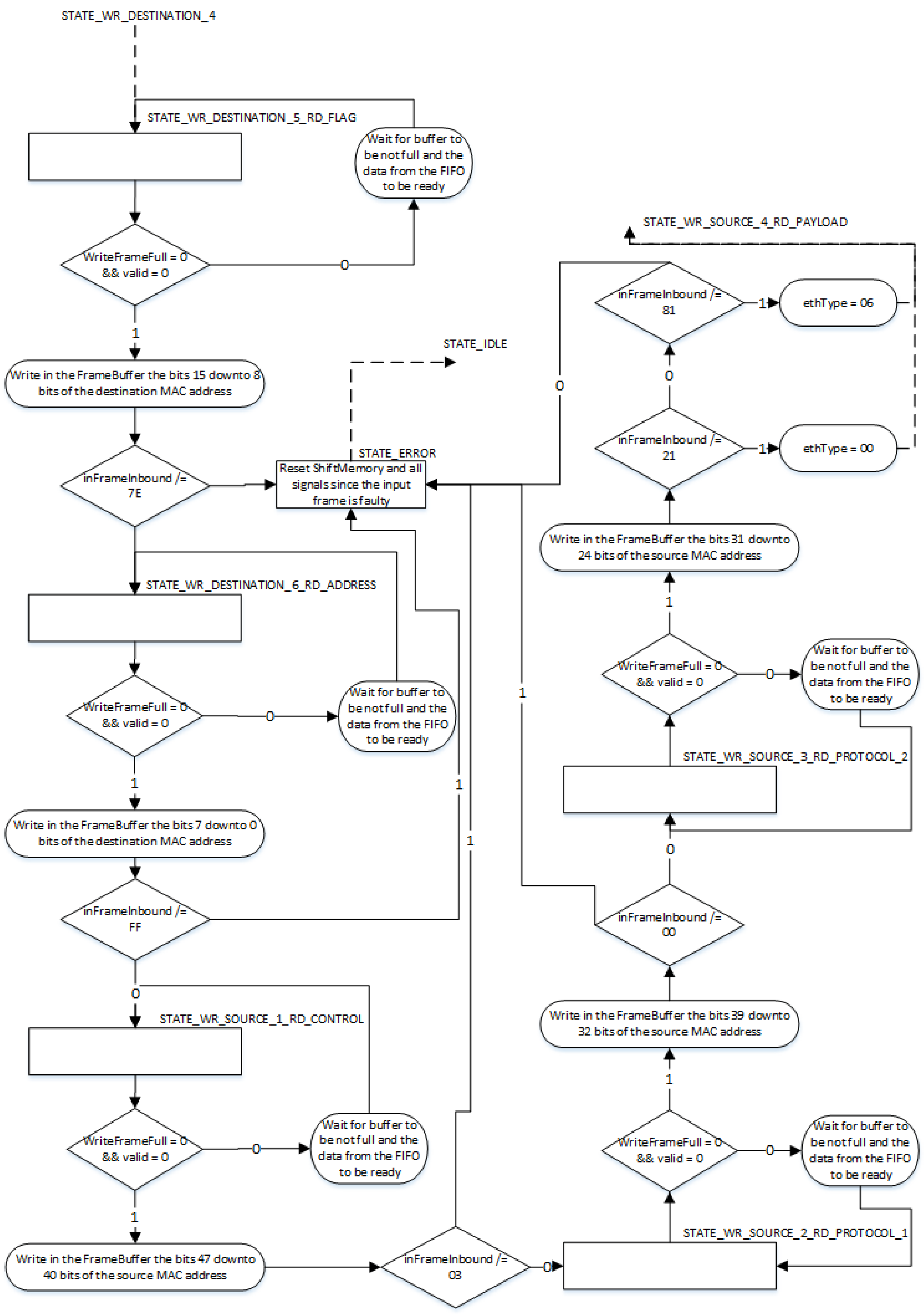

STATE_WR_DESTINATION_4
STATE_WR_DESTINATION_5_RD_FLAG
Wait for buffer to be not full and the data from the FIFO to be ready
WriteFrameFull = 0 && valid = 0
0
1
Write in the FrameBuffer the bits 15 downto 8 bits of the destination MAC address
STATE_IDLE
STATE_ERROR
Reset ShiftMemory and all signals since the input frame is faulty
inFrameInbound /= 7E
STATE_WR_DESTINATION_6_RD_ADDRESS
WriteFrameFull = 0 && valid = 0
0
Wait for buffer to be not full and the data from the FIFO to be ready
1
Write in the FrameBuffer the bits 7 downto 0 bits of the destination MAC address
inFrameInbound /= FF
0
STATE_WR_SOURCE_1_RD_CONTROL
WriteFrameFull = 0 && valid = 0
0
Wait for buffer to be not full and the data from the FIFO to be ready
1
Write in the FrameBuffer the bits 47 downto 40 bits of the source MAC address
inFrameInbound /= 03
0
1
1
1
STATE_WR_SOURCE_4_RD_PAYLOAD
inFrameInbound /= 81
1
ethType = 06
0
0
inFrameInbound /= 21
1
ethType = 00
Write in the FrameBuffer the bits 31 downto 24 bits of the source MAC address
1
WriteFrameFull = 0 && valid = 0
0
Wait for buffer to be not full and the data from the FIFO to be ready
STATE_WR_SOURCE_3_RD_PROTOCOL_2
0
inFrameInbound /= 00
Write in the FrameBuffer the bits 39 downto 32 bits of the source MAC address
1
WriteFrameFull = 0 && valid = 0
0
Wait for buffer to be not full and the data from the FIFO to be ready
STATE_WR_SOURCE_2_RD_PROTOCOL_1

Figure 14.2: Control flow chart, behavioral representation. Part 2 of the Preamble in the inbound part

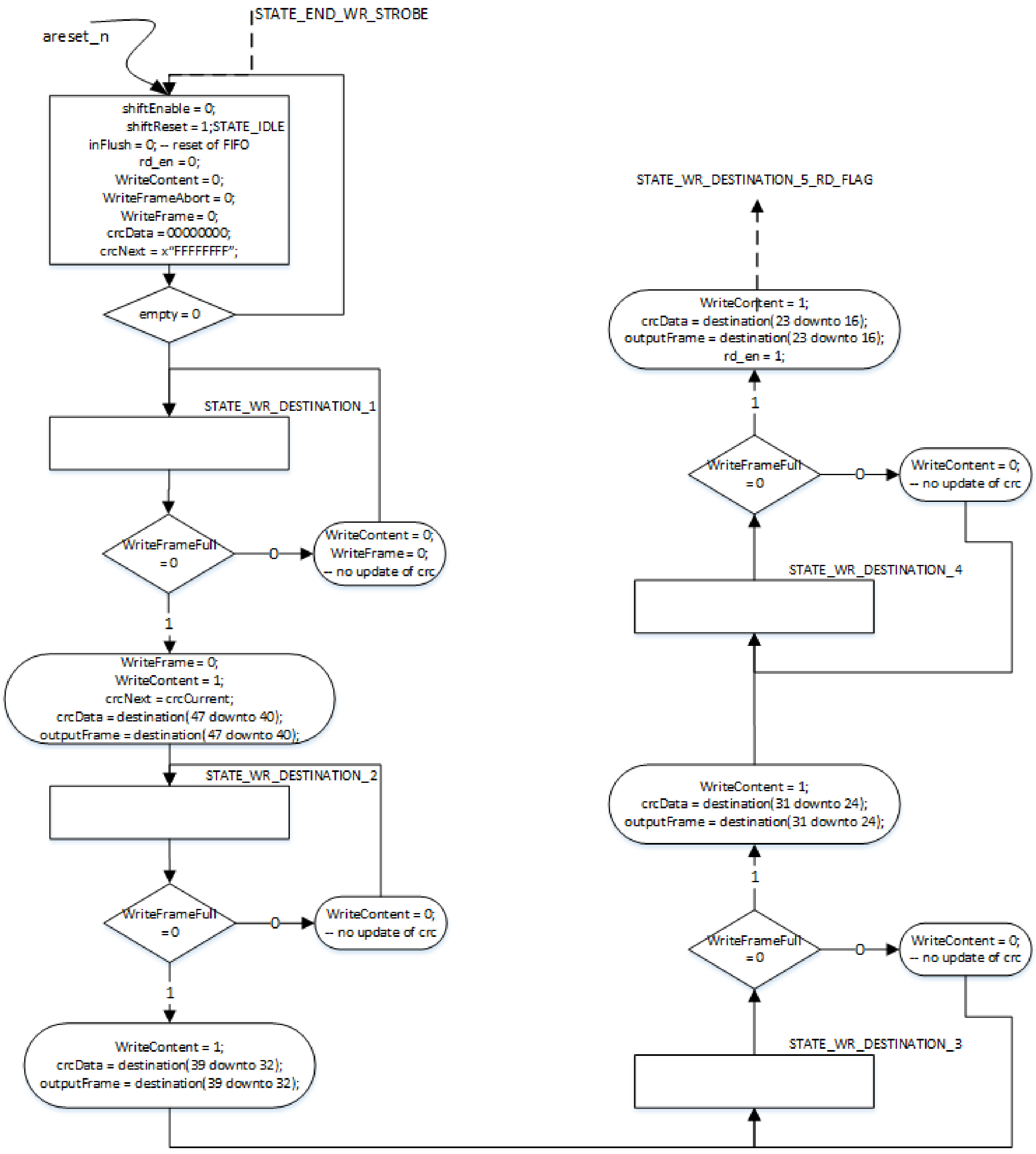
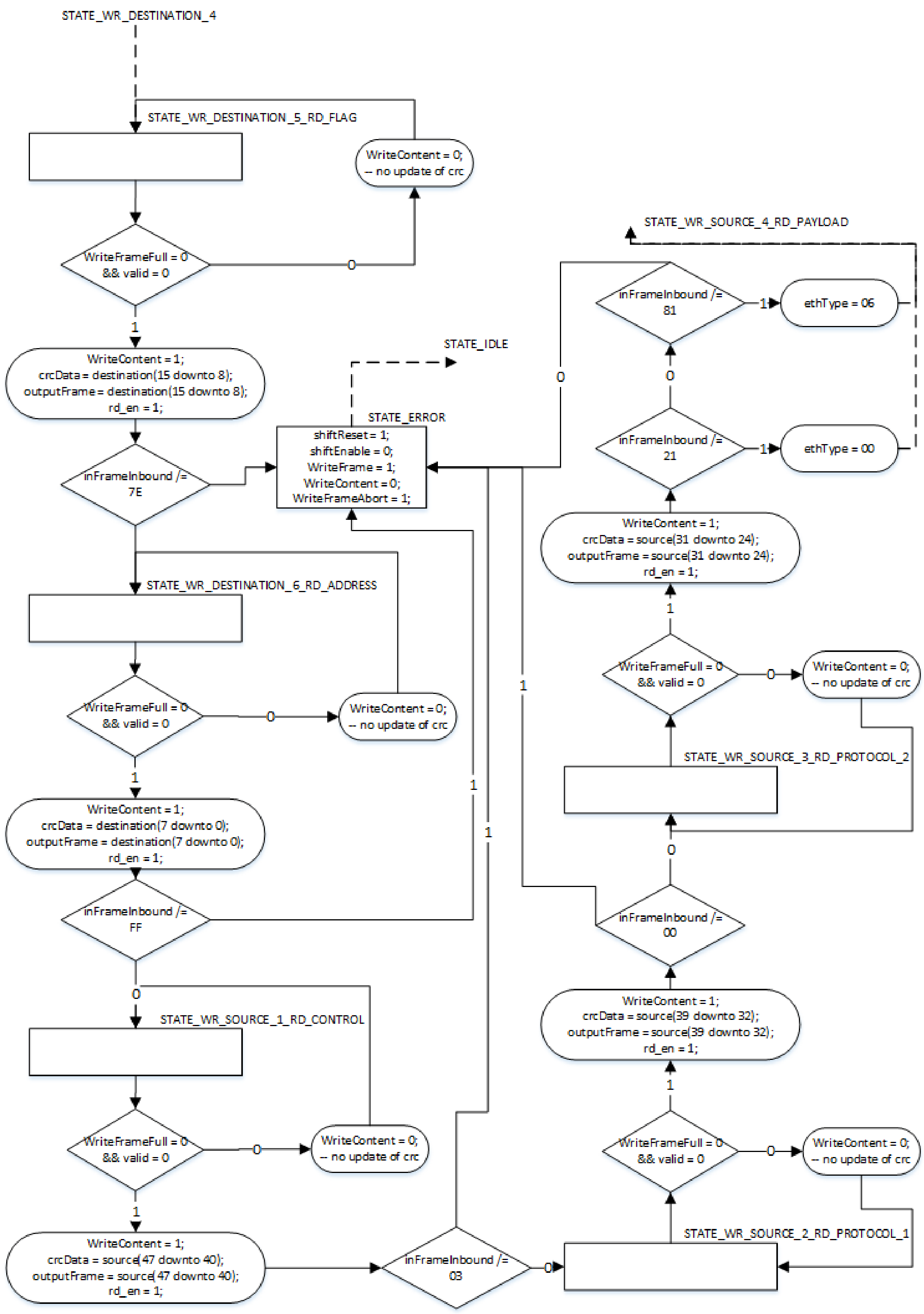

areset_n
STATE_END_WR_STROBE
shiftEnable = 0;
shiftReset = 1;STATE_IDLE
inFlush = 0; -- reset of FIFO
rd_en = 0;
WriteContent = 0;
WriteFrameAbort = 0;
WriteFrame = 0;
crcData = 00000000;
crcNext = x"FFFFFFFF";
empty = 0
STATE_WR_DESTINATION_1
WriteFrameFull = 0
0
WriteContent = 0; WriteFrame = 0; -- no update of crc
1
WriteFrame = 0; WriteContent = 1; crcNext = crcCurrent; crcData = destination(47 downto 40); outputFrame = destination(47 downto 40);
STATE_WR_DESTINATION_2
WriteFrameFull = 0
0
WriteContent = 0; -- no update of crc
1
WriteContent = 1; crcData = destination(39 downto 32); outputFrame = destination(39 downto 32);
STATE_WR_DESTINATION_3
WriteFrameFull = 0
0
WriteContent = 0; -- no update of crc
1
WriteContent = 1; crcData = destination(31 downto 24); outputFrame = destination(31 downto 24);
STATE_WR_DESTINATION_4
WriteFrameFull = 0
0
WriteContent = 0; -- no update of crc
1
WriteContent = 1; crcData = destination(23 downto 16); outputFrame = destination(23 downto 16); rd_en = 1;
STATE_WR_DESTINATION_5_RD_FLAG

Figure 14.3: Control flow chart, explicit controls. Part 1 of the Preamble in the inbound part

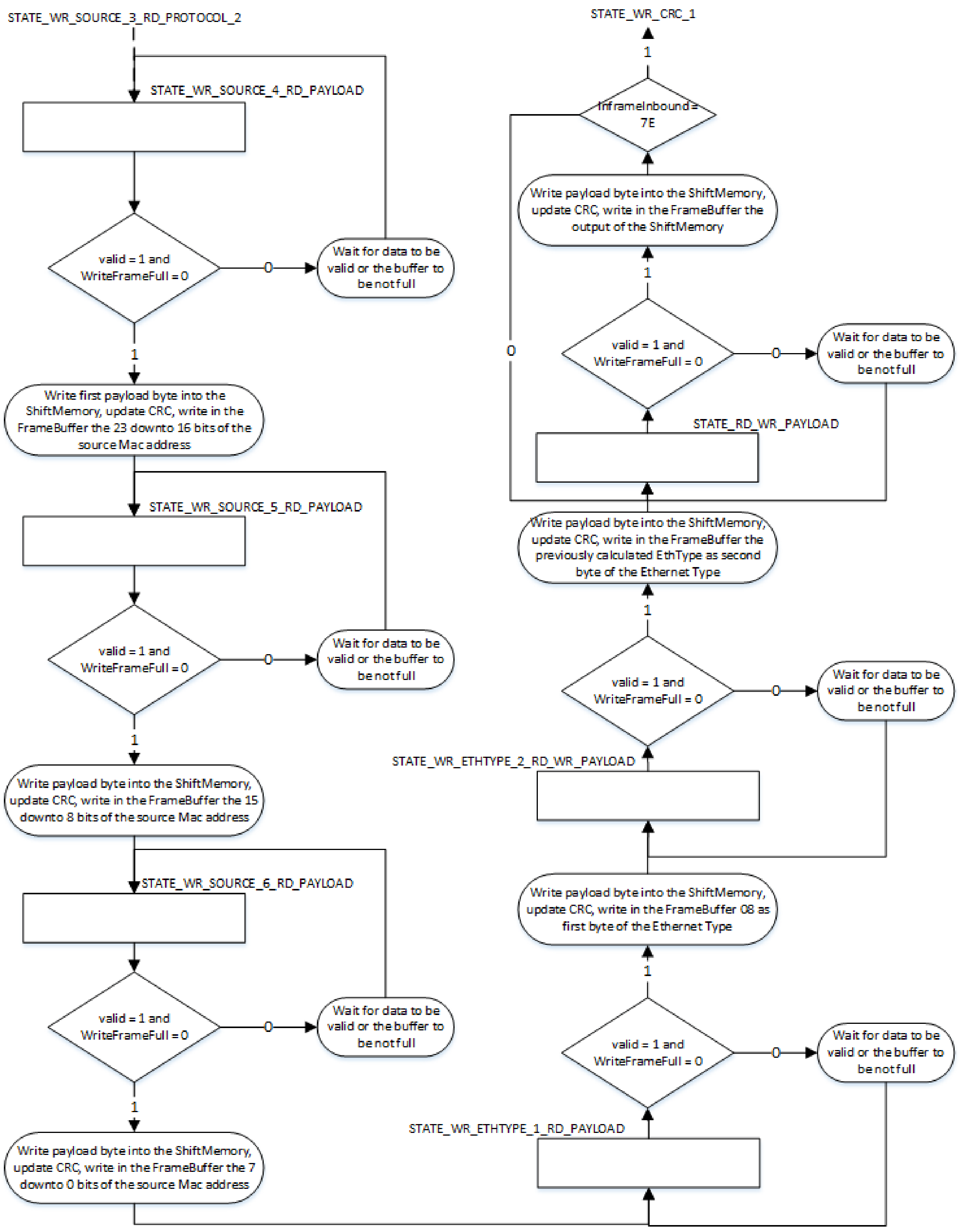


STATE_WR_DESTINATION_4
STATE_WR_DESTINATION_5_RD_FLAG
WriteContent = 0; -- no update of crc
WriteFrameFull = 0 && valid = 0
0
1
WriteContent = 1; crcData = destination(15 downto 8); outputFrame = destination(15 downto 8); rd_en = 1;
inFrameInbound /= 7E
STATE_IDLE
STATE_ERROR
shiftReset = 1; shiftEnable = 0; WriteFrame = 1; WriteContent = 0; WriteFrameAbort = 1;
STATE_WR_DESTINATION_6_RD_ADDRESS
WriteFrameFull = 0 && valid = 0
WriteContent = 0; -- no update of crc
WriteContent = 1; crcData = destination(7 downto 0); outputFrame = destination(7 downto 0); rd_en = 1;
inFrameInbound /= FF
STATE_WR_SOURCE_1_RD_CONTROL
WriteFrameFull = 0 && valid = 0
WriteContent = 0; -- no update of crc
WriteContent = 1; crcData = source(47 downto 40); outputFrame = source(47 downto 40); rd_en = 1;
inFrameInbound /= 03
STATE_WR_SOURCE_2_RD_PROTOCOL_1
WriteFrameFull = 0 && valid = 0
WriteContent = 0; -- no update of crc
WriteContent = 1; crcData = source(39 downto 32); outputFrame = source(39 downto 32); rd_en = 1;
inFrameInbound /= 00
STATE_WR_SOURCE_3_RD_PROTOCOL_2
WriteFrameFull = 0 && valid = 0
WriteContent = 0; -- no update of crc
WriteContent = 1; crcData = source(31 downto 24); outputFrame = source(31 downto 24); rd_en = 1;
inFrameInbound /= 21
ethType = 00
inFrameInbound /= 81
ethType = 06
STATE_WR_SOURCE_4_RD_PAYLOAD

Figure 14.4: Control flow chart, explicit controls. Part 2 of the Preamble in the inbound part

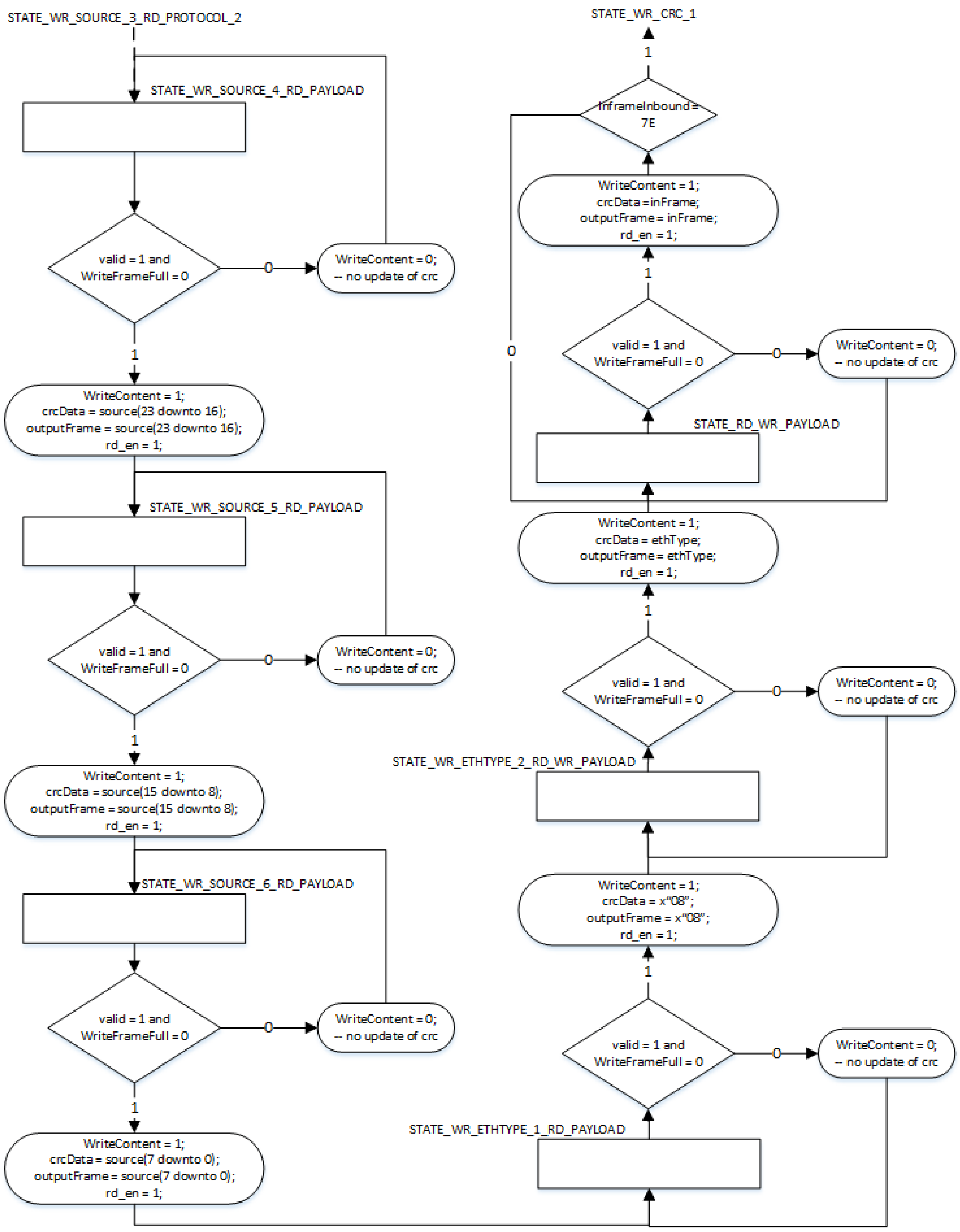
14.1.2 Payload

Figures 14.5 and 14.6 show the control flow charts of the Payload part. They feature the behavior and explicit controls part, respectively.



STATE_WR_SOURCE_3_RD_PROTOCOL_2
STATE_WR_SOURCE_4_RD_PAYLOAD
valid = 1 and WriteFrameFull = 0
Wait for data to be valid or the buffer to be not full
Write first payload byte into the ShiftMemory, update CRC, write in the FrameBuffer the 23 downto 16 bits of the source Mac address
STATE_WR_SOURCE_5_RD_PAYLOAD
Write payload byte into the ShiftMemory, update CRC, write in the FrameBuffer the 15 downto 8 bits of the source Mac address
STATE_WR_SOURCE_6_RD_PAYLOAD
Write payload byte into the ShiftMemory, update CRC, write in the FrameBuffer the 7 downto 0 bits of the source Mac address
STATE_WR_CRC_1
InframeInbound = 7E
Write payload byte into the ShiftMemory, update CRC, write in the FrameBuffer the output of the ShiftMemory
STATE_RD_WR_PAYLOAD
Write payload byte into the ShiftMemory, update CRC, write in the FrameBuffer the previously calculated EthType as second byte of the Ethernet Type
STATE_WR_ETHTYPE_2_RD_WR_PAYLOAD
Write payload byte into the ShiftMemory, update CRC, write in the FrameBuffer 08 as first byte of the Ethernet Type
STATE_WR_ETHTYPE_1_RD_PAYLOAD
0
1

Figure 14.5: Control flow chart, behavioral representation. Payload management in the inbound part



STATE_WR_SOURCE_3_RD_PROTOCOL_2
STATE_WR_SOURCE_4_RD_PAYLOAD
valid = 1 and WriteFrameFull = 0
0
WriteContent = 0; -- no update of crc
1
WriteContent = 1; crcData = source(23 downto 16); outputFrame = source(23 downto 16); rd_en = 1;
STATE_WR_SOURCE_5_RD_PAYLOAD
valid = 1 and WriteFrameFull = 0
WriteContent = 0; -- no update of crc
WriteContent = 1; crcData = source(15 downto 8); outputFrame = source(15 downto 8); rd_en = 1;
STATE_WR_SOURCE_6_RD_PAYLOAD
valid = 1 and WriteFrameFull = 0
WriteContent = 0; -- no update of crc
WriteContent = 1; crcData = source(7 downto 0); outputFrame = source(7 downto 0); rd_en = 1;
STATE_WR_CRC_1
inframeInbound = 7E
WriteContent = 1; crcData = inFrame; outputFrame = inFrame; rd_en = 1;
STATE_RD_WR_PAYLOAD
WriteContent = 1; crcData = ethType; outputFrame = ethType; rd_en = 1;
STATE_WR_ETHTYPE_2_RD_WR_PAYLOAD
WriteContent = 1; crcData = x"08"; outputFrame = x"08"; rd_en = 1;
STATE_WR_ETHTYPE_1_RD_PAYLOAD

Figure 14.6: Control flow chart, explicit controls. Payload management in the inbound part

14.2.3 CRC and Ending

In Figures 14.7 and 14.8 are shown the control flow charts of the last bytes of CRC and frame ending part. Respectively, in the former there is the behavior and in the latter the explicit controls used in each state.

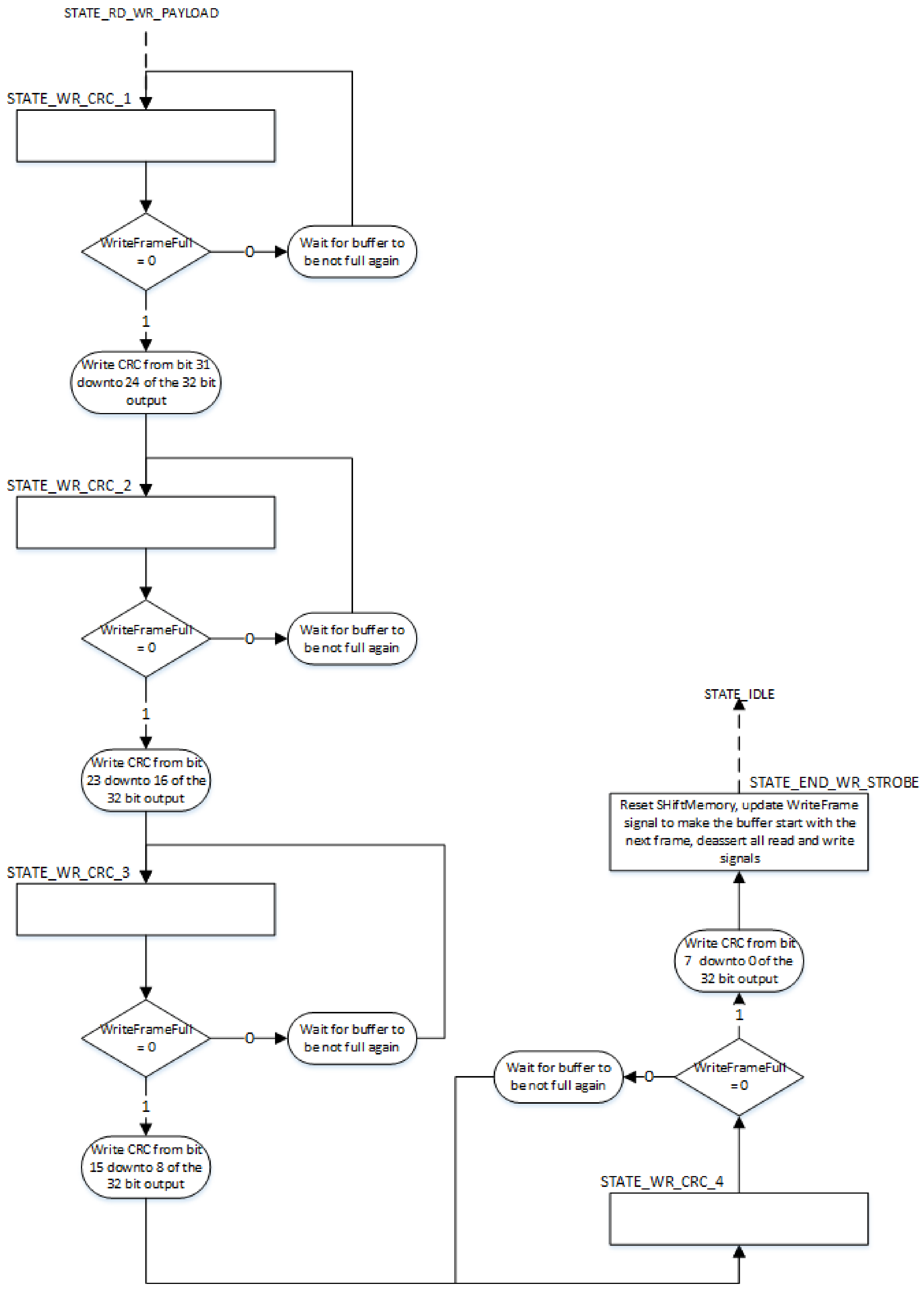
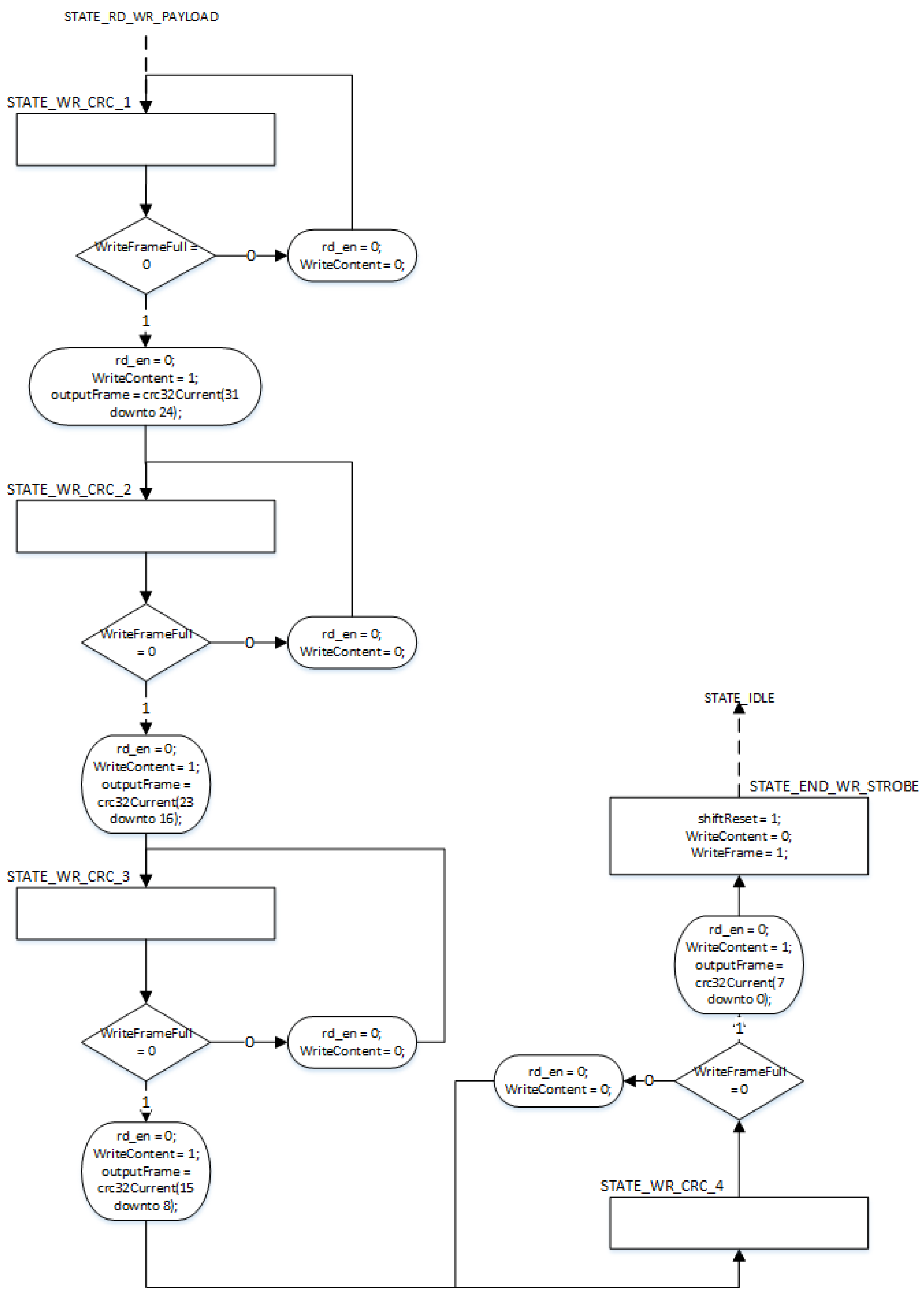

STATE_RD_WR_PAYLOAD
STATE_WR_CRC_1
WriteFrameFull = 0
0
Wait for buffer to be not full again
1
Write CRC from bit 31 downto 24 of the 32 bit output
STATE_WR_CRC_2
WriteFrameFull = 0
0
Wait for buffer to be not full again
1
Write CRC from bit 23 downto 16 of the 32 bit output
STATE_WR_CRC_3
WriteFrameFull = 0
0
Wait for buffer to be not full again
1
Write CRC from bit 15 downto 8 of the 32 bit output
STATE_IDLE
STATE_END_WR_STROBE
Reset SHiftMemory, update WriteFrame signal to make the buffer start with the next frame, deassert all read and write signals
Write CRC from bit 7 downto 0 of the 32 bit output
1
WriteFrameFull = 0
0
Wait for buffer to be not full again
STATE_WR_CRC_4

Figure 14.7: Control flow chart, behavioral representation. CRC and frame end management in the inbound part



STATE_RD_WR_PAYLOAD
STATE_WR_CRC_1
WriteFrameFull = 0
0
rd_en = 0; WriteContent = 0;
1
rd_en = 0; WriteContent = 1; outputFrame = crc32Current(31 downto 24);
STATE_WR_CRC_2
WriteFrameFull = 0
0
rd_en = 0; WriteContent = 0;
1
rd_en = 0; WriteContent = 1; outputFrame = crc32Current(23 downto 16);
STATE_WR_CRC_3
WriteFrameFull = 0
0
rd_en = 0; WriteContent = 0;
1
rd_en = 0; WriteContent = 1; outputFrame = crc32Current(15 downto 8);
STATE_IDLE
STATE_END_WR_STROBE
shiftReset = 1; WriteContent = 0; WriteFrame = 1;
rd_en = 0; WriteContent = 1; outputFrame = crc32Current(7 downto 0);
'1'
WriteFrameFull = 0
0
rd_en = 0; WriteContent = 0;
STATE_WR_CRC_4

Figure 14.8: Control flow chart, explicit controls. CRC and frame end management in the inbound part

14.2 Outbound Control Flow charts

This section follows the same structure of 14.1, but all the charts are referred to the outbound part.

14.2.1 Preamble

In Figures 14.9 and 14.10 are shown respectively the start of the FSM together with the reading of the destination MAC address and the start of the writing part of the PPP frame. In Figures 14.11 and 14.12 are represented the same charts but with the explicit controls used in every state.

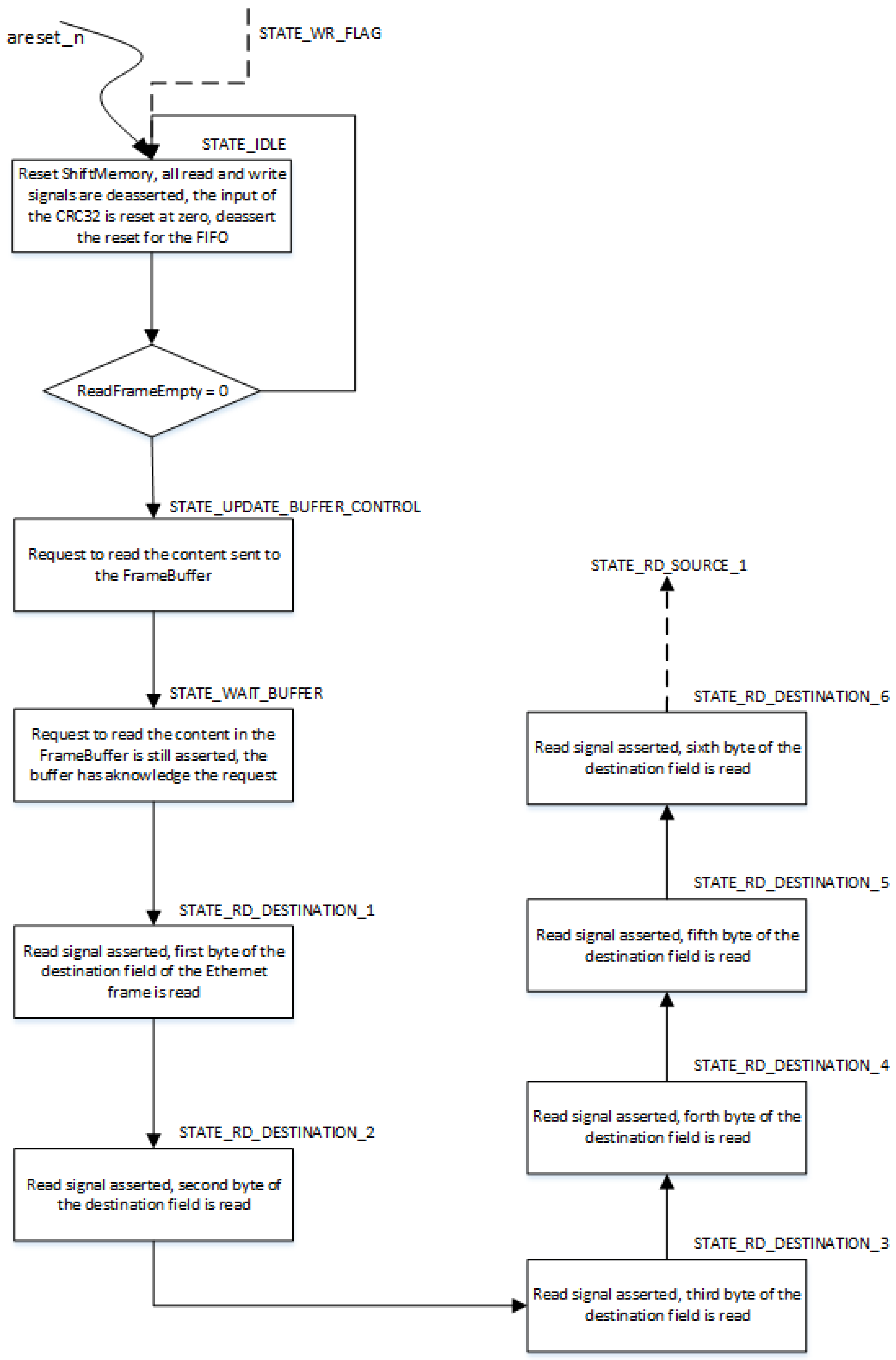

areset_n
STATE_WR_FLAG
STATE_IDLE
Reset ShiftMemory, all read and write signals are deasserted, the input of the CRC32 is reset at zero, deassert the reset for the FIFO
ReadFrameEmpty = 0
STATE_UPDATE_BUFFER_CONTROL
Request to read the content sent to the FrameBuffer
STATE_WAIT_BUFFER
Request to read the content in the FrameBuffer is still asserted, the buffer has aknowledge the request
STATE_RD_DESTINATION_1
Read signal asserted, first byte of the destination field of the Ethernet frame is read
STATE_RD_DESTINATION_2
Read signal asserted, second byte of the destination field is read
STATE_RD_DESTINATION_3
Read signal asserted, third byte of the destination field is read
STATE_RD_DESTINATION_4
Read signal asserted, forth byte of the destination field is read
STATE_RD_DESTINATION_5
Read signal asserted, fifth byte of the destination field is read
STATE_RD_DESTINATION_6
Read signal asserted, sixth byte of the destination field is read
STATE_RD_SOURCE_1

Figure 14.9: Control flow chart, behavioral representation. Part 1 of the Preamble in the outbound part

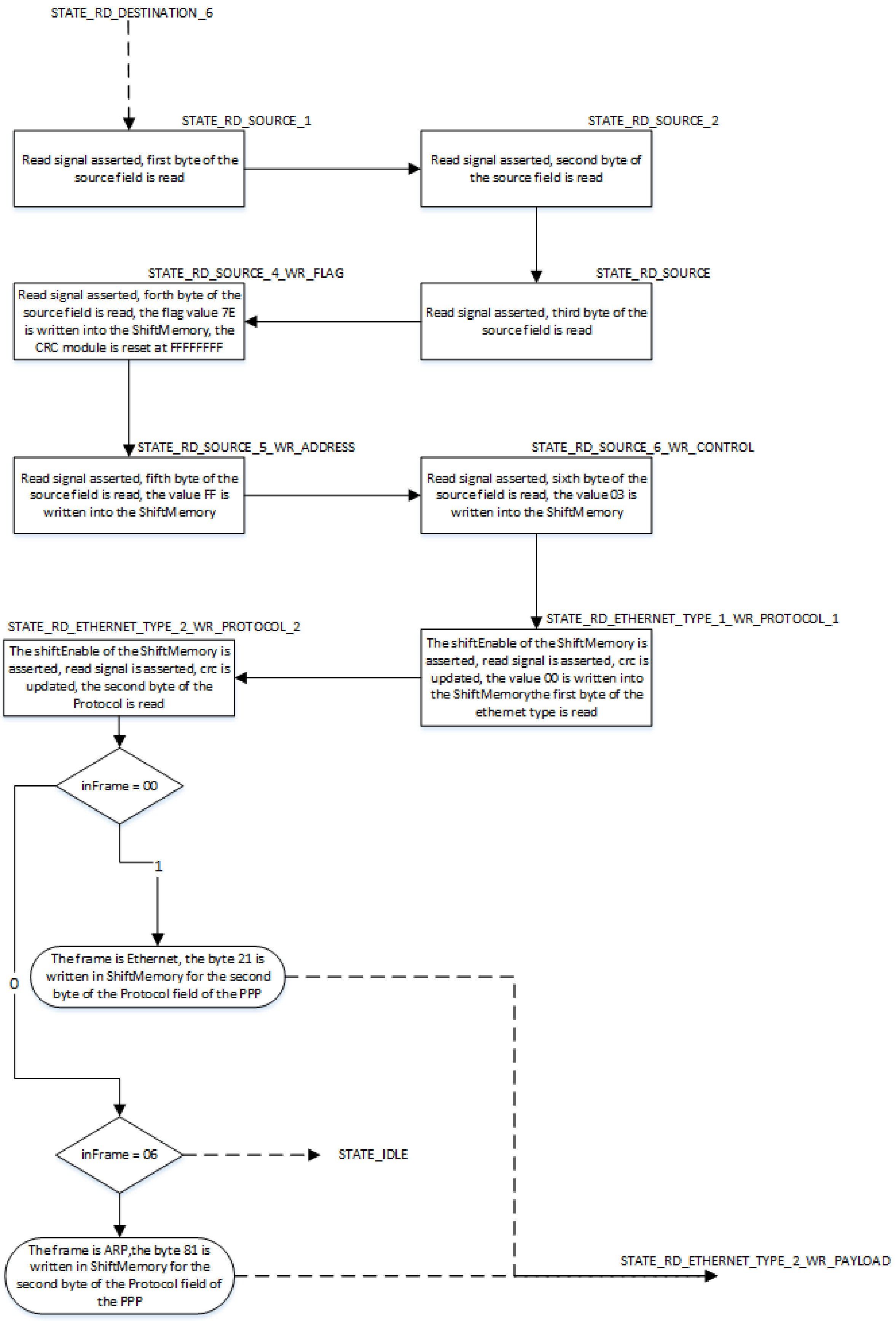

STATE_RD_DESTINATION_6
STATE_RD_SOURCE_1
Read signal asserted, first byte of the source field is read
STATE_RD_SOURCE_2
Read signal asserted, second byte of the source field is read
STATE_RD_SOURCE_4_WR_FLAG
Read signal asserted, forth byte of the source field is read, the flag value 7E is written into the ShiftMemory, the CRC module is reset at FFFFFFFF
STATE_RD_SOURCE
Read signal asserted, third byte of the source field is read
STATE_RD_SOURCE_5_WR_ADDRESS
Read signal asserted, fifth byte of the source field is read, the value FF is written into the ShiftMemory
STATE_RD_SOURCE_6_WR_CONTROL
Read signal asserted, sixth byte of the source field is read, the value 03 is written into the ShiftMemory
STATE_RD_ETHERNET_TYPE_2_WR_PROTOCOL_2
The shiftEnable of the ShiftMemory is asserted, read signal is asserted, crc is updated, the second byte of the Protocol is read
STATE_RD_ETHERNET_TYPE_1_WR_PROTOCOL_1
The shiftEnable of the ShiftMemory is asserted, read signal is asserted, crc is updated, the value 00 is written into the ShiftMemorythe first byte of the ethernet type is read
inFrame = 00
1
0
The frame is Ethernet, the byte 21 is written in ShiftMemory for the second byte of the Protocol field of the PPP
inFrame = 06
STATE_IDLE
The frame is ARP,the byte 81 is written in ShiftMemory for the second byte of the Protocol field of the PPP
STATE_RD_ETHERNET_TYPE_2_WR_PAYLOAD

Figure 14.10: Control flow chart, behavioral representation. Part 2 of the Preamble in the outbound part

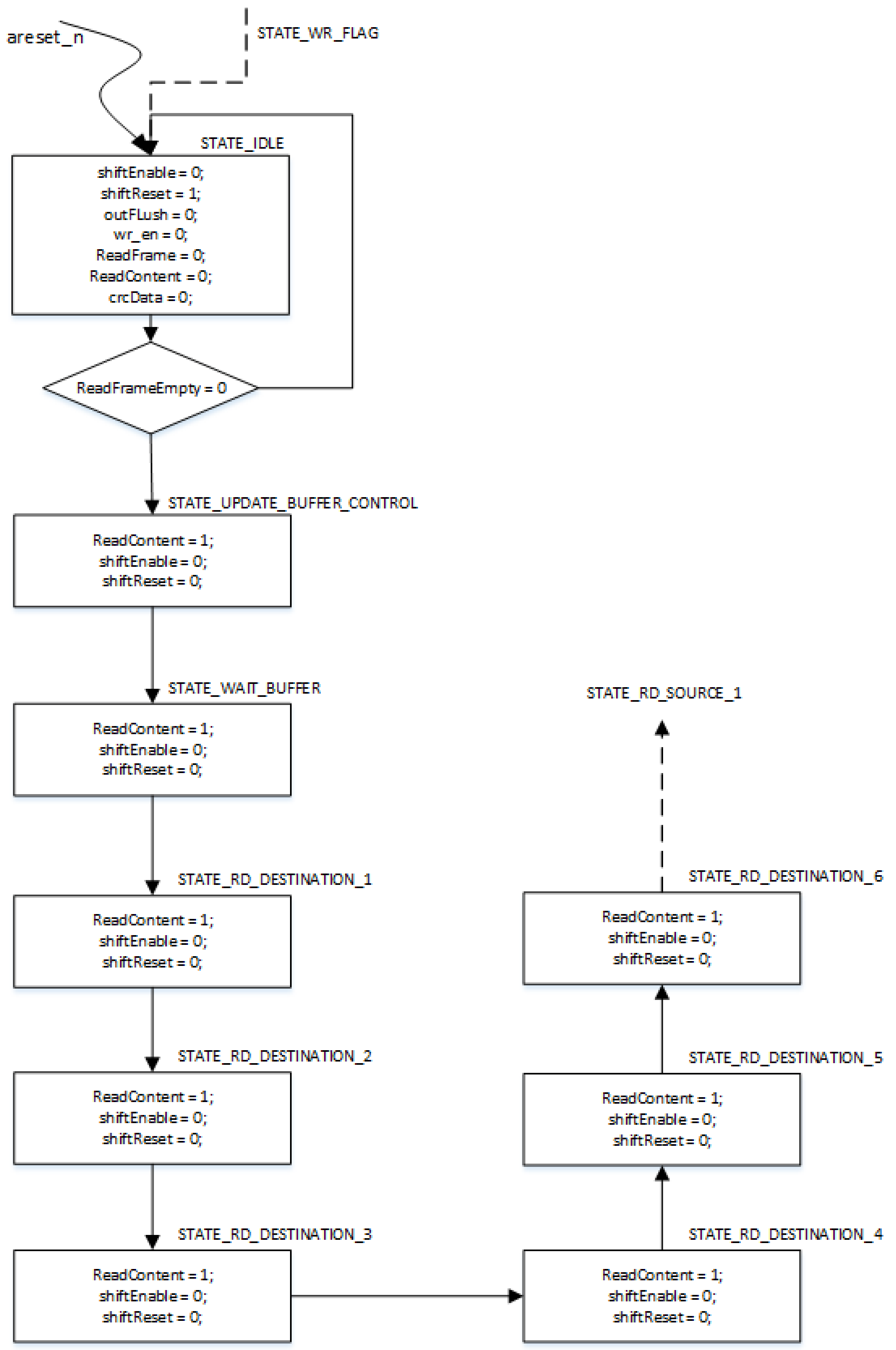


Figure 14.11: Control flow chart, explicit controls. Part 1 of the Preamble in the outbound part

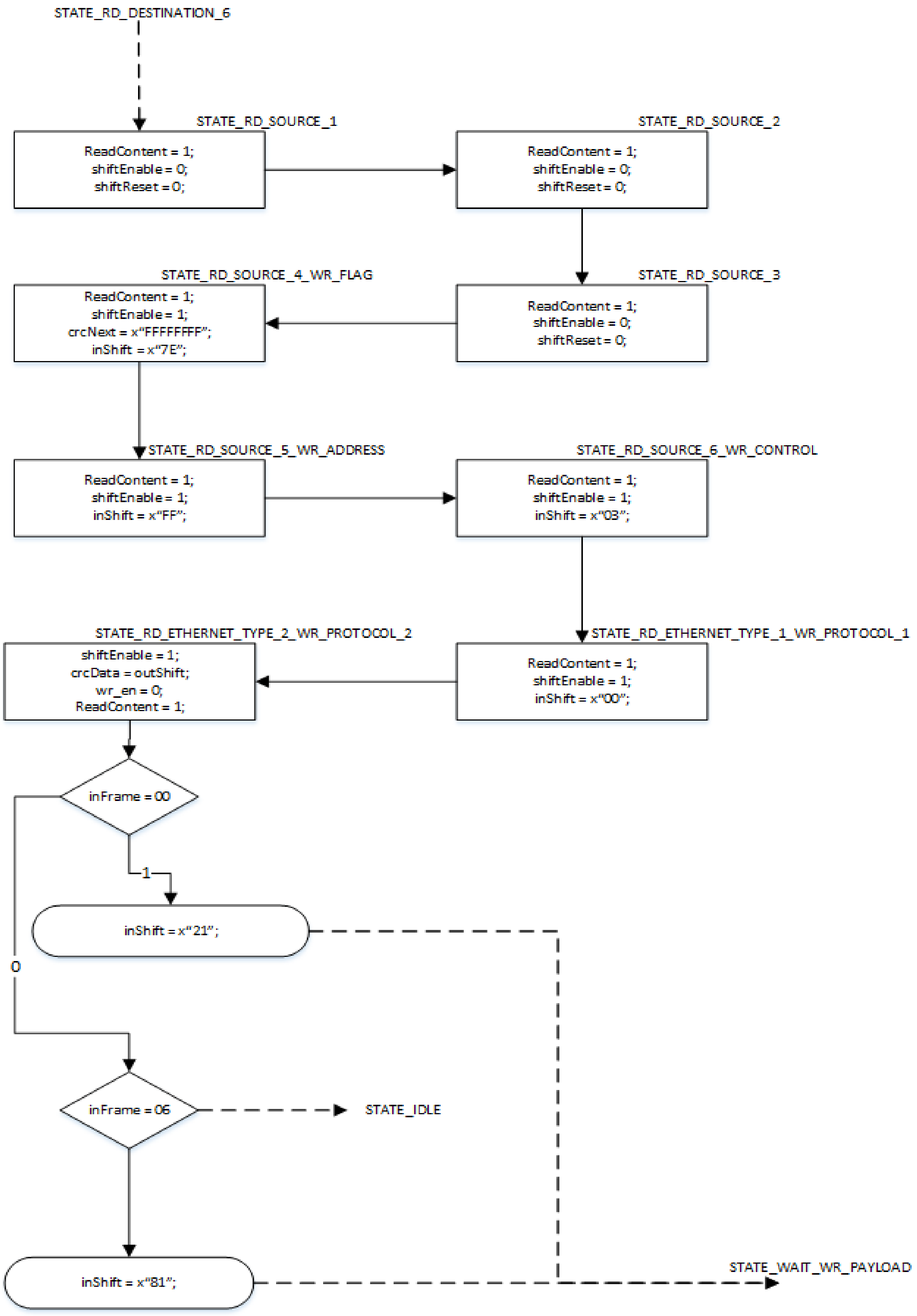

STATE_RD_DESTINATION_6
STATE_RD_SOURCE_1
ReadContent = 1; shiftEnable = 0; shiftReset = 0;
STATE_RD_SOURCE_2
ReadContent = 1; shiftEnable = 0; shiftReset = 0;
STATE_RD_SOURCE_3
ReadContent = 1; shiftEnable = 0; shiftReset = 0;
STATE_RD_SOURCE_4_WR_FLAG
ReadContent = 1; shiftEnable = 1; crcNext = x"FFFFFFFF"; inShift = x"7E";
STATE_RD_SOURCE_5_WR_ADDRESS
ReadContent = 1; shiftEnable = 1; inShift = x"FF";
STATE_RD_SOURCE_6_WR_CONTROL
ReadContent = 1; shiftEnable = 1; inShift = x"03";
STATE_RD_ETHERNET_TYPE_1_WR_PROTOCOL_1
ReadContent = 1; shiftEnable = 1; inShift = x"00";
STATE_RD_ETHERNET_TYPE_2_WR_PROTOCOL_2
shiftEnable = 1; crcData = outShift; wr_en = 0; ReadContent = 1;
inFrame = 00
1
0
inShift = x"21";
inFrame = 06
STATE_IDLE
inShift = x"81";
STATE_WAIT_WR_PAYLOAD

Figure 14.12: Control flow chart, explicit controls. Part 2 of the Preamble in the outbound part

14.2.2 Payload

In Figures 14.13 and 14.14 are shown the control flow charts of the Payload part. In the former, there is the behavior , in the latter, the explicit controls used in each state.

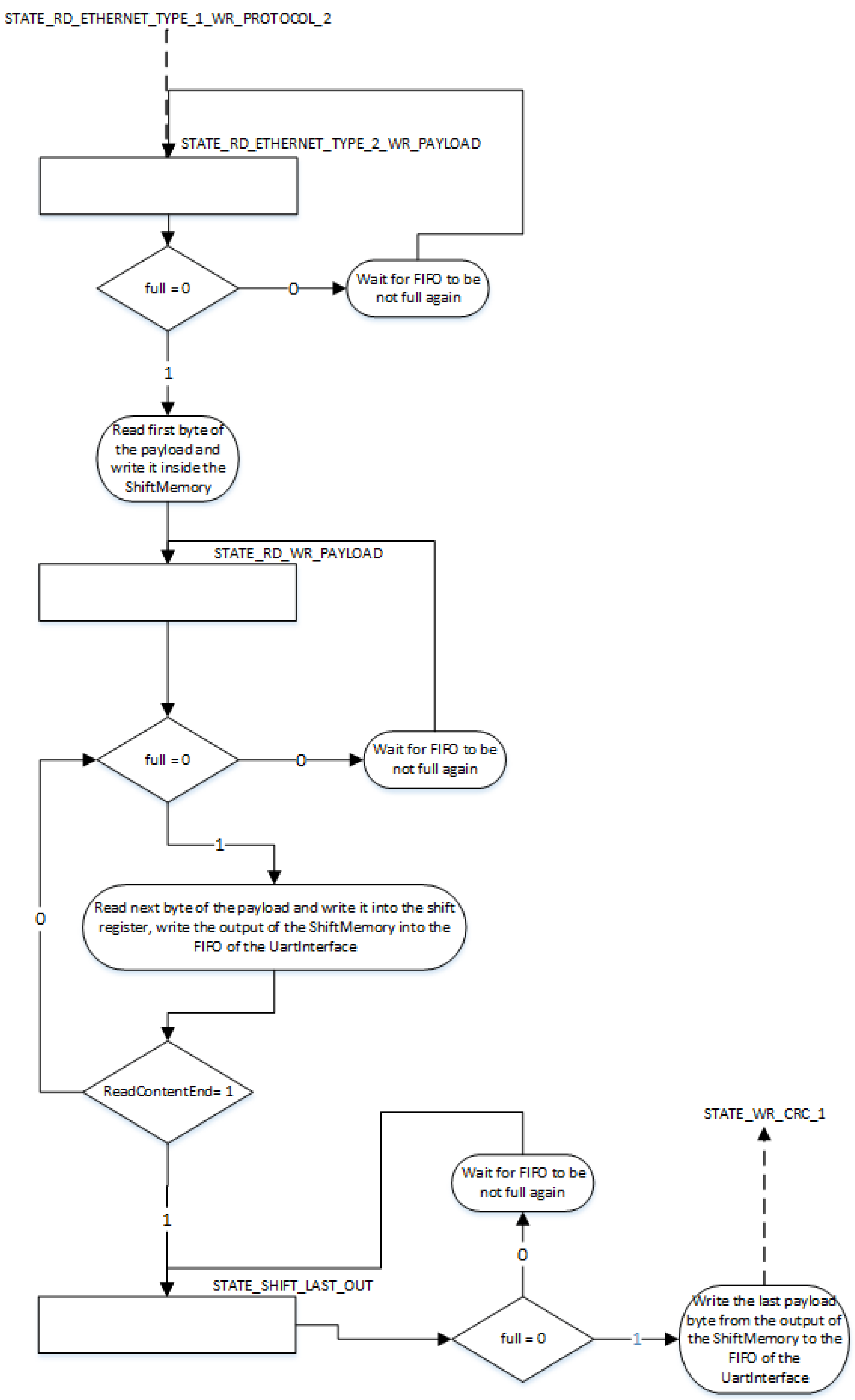
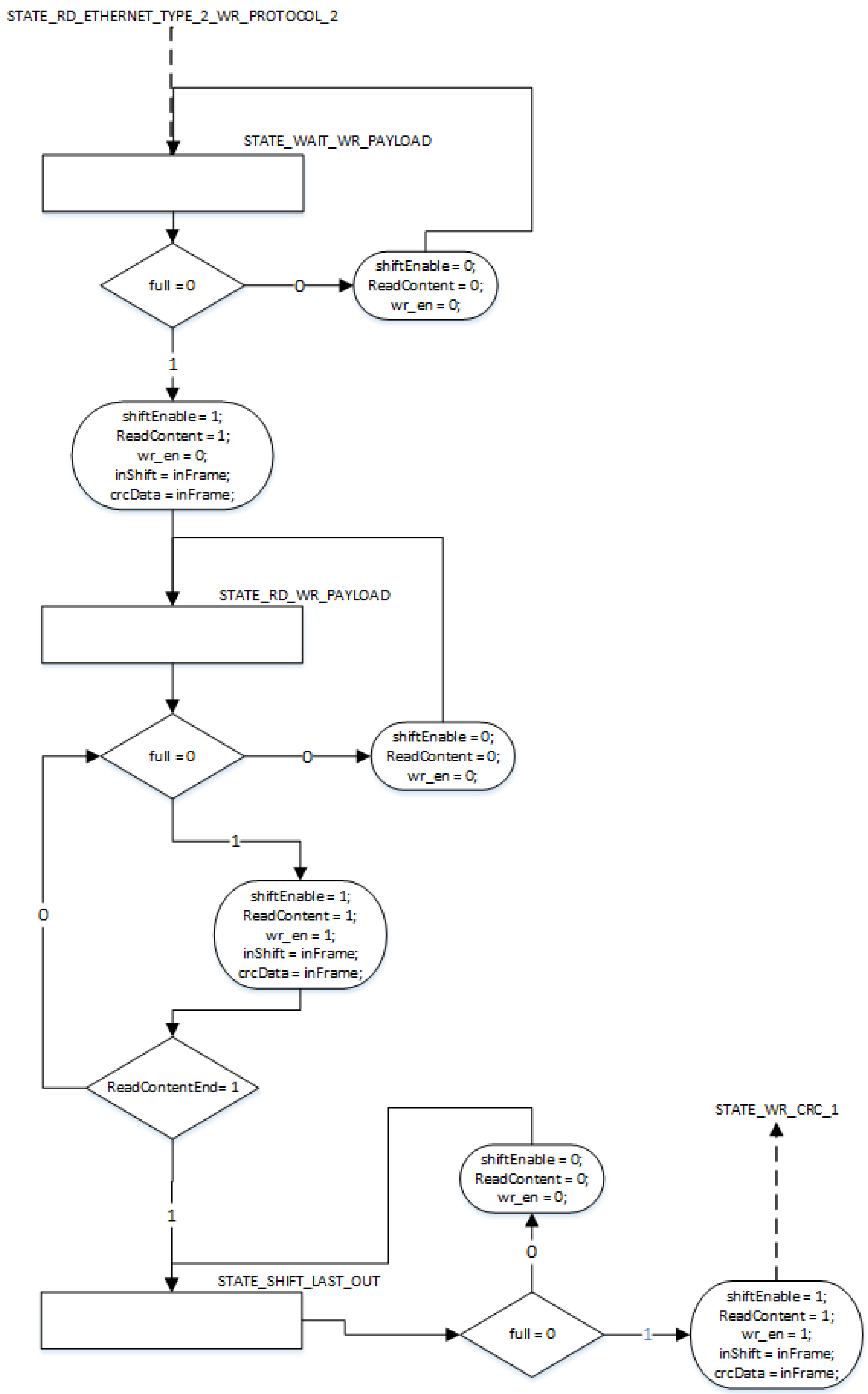

STATE_RD_ETHERNET_TYPE_1_WR_PROTOCOL_2
STATE_RD_ETHERNET_TYPE_2_WR_PAYLOAD
full = 0
0
Wait for FIFO to be not full again
1
Read first byte of the payload and write it inside the ShiftMemory
STATE_RD_WR_PAYLOAD
full = 0
0
Wait for FIFO to be not full again
1
Read next byte of the payload and write it into the shift register, write the output of the ShiftMemory into the FIFO of the UartInterface
0
ReadContentEnd= 1
1
STATE_SHIFT_LAST_OUT
Wait for FIFO to be not full again
0
full = 0
1
Write the last payload byte from the output of the ShiftMemory to the FIFO of the UartInterface
STATE_WR_CRC_1

Figure 14.13: Control flow chart, behavioral representation. Payload management in the outbound part

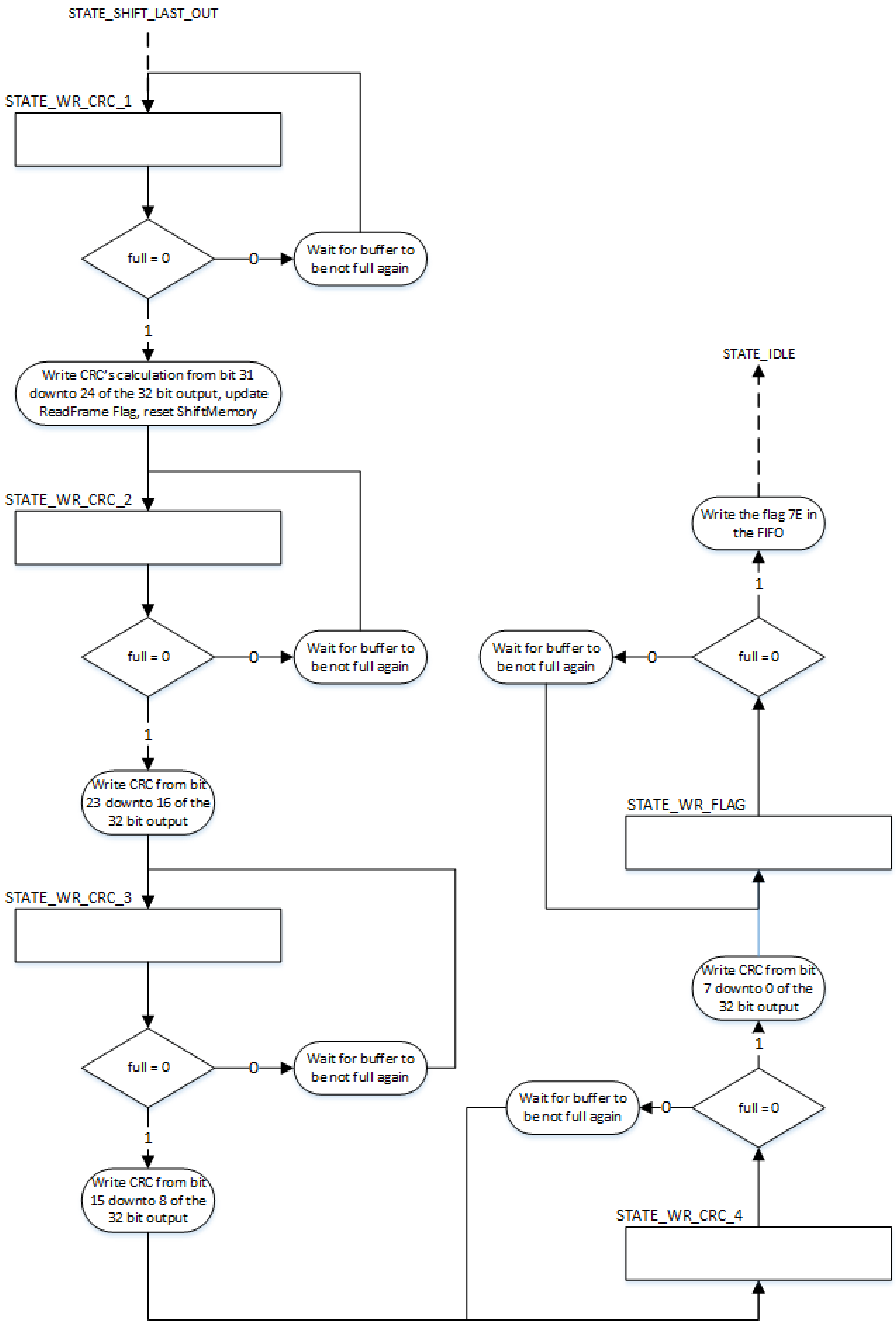


STATE_RD_ETHERNET_TYPE_2_WR_PROTOCOL_2
STATE_WAIT_WR_PAYLOAD
full = 0
0
shiftEnable = 0; ReadContent = 0; wr_en = 0;
1
shiftEnable = 1; ReadContent = 1; wr_en = 0; inShift = inFrame; crcData = inFrame;
STATE_RD_WR_PAYLOAD
full = 0
0
shiftEnable = 0; ReadContent = 0; wr_en = 0;
1
shiftEnable = 1; ReadContent = 1; wr_en = 1; inShift = inFrame; crcData = inFrame;
0
ReadContentEnd= 1
1
STATE_SHIFT_LAST_OUT
full = 0
0
shiftEnable = 0; ReadContent = 0; wr_en = 0;
1
shiftEnable = 1; ReadContent = 1; wr_en = 1; inShift = inFrame; crcData = inFrame;
STATE_WR_CRC_1

Figure 14.14: Control flow chart, explicit controls. Payload management in the outbound part

14.2.3 CRC and Ending

Figures 14.15 and 14.16 show the control flow charts of the last bytes of CRC and frame ending part. In the former, there is the behavior and, in the latter, the explicit controls used in each state.



STATE_SHIFT_LAST_OUT
STATE_WR_CRC_1
full = 0
0
Wait for buffer to be not full again
1
Write CRC's calculation from bit 31 downto 24 of the 32 bit output, update ReadFrame Flag, reset ShiftMemory
STATE_WR_CRC_2
full = 0
0
Wait for buffer to be not full again
1
Write CRC from bit 23 downto 16 of the 32 bit output
STATE_WR_CRC_3
full = 0
0
Wait for buffer to be not full again
1
Write CRC from bit 15 downto 8 of the 32 bit output
STATE_IDLE
Write the flag 7E in the FIFO
1
Wait for buffer to be not full again
0
full = 0
STATE_WR_FLAG
Write CRC from bit 7 downto 0 of the 32 bit output
1
Wait for buffer to be not full again
0
full = 0
STATE_WR_CRC_4

Figure 14.15: Control flow chart, behavioral representation. CRC and frame end management in the outbound part

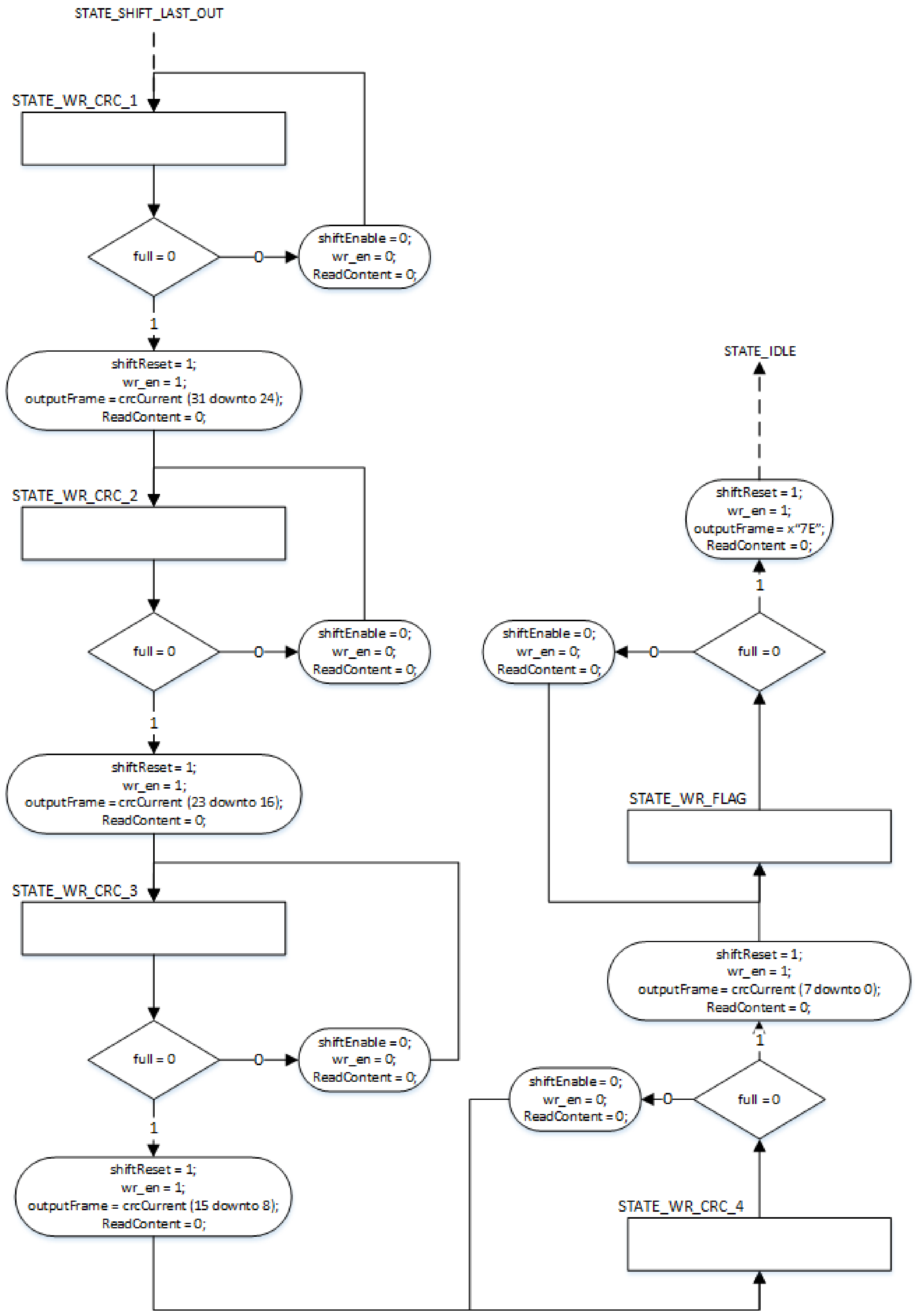

STATE_SHIFT_LAST_OUT
STATE_WR_CRC_1
full = 0
0
shiftEnable = 0; wr_en = 0; ReadContent = 0;
1
shiftReset = 1; wr_en = 1; outputFrame = crcCurrent (31 downto 24); ReadContent = 0;
STATE_WR_CRC_2
full = 0
0
shiftEnable = 0; wr_en = 0; ReadContent = 0;
1
shiftReset = 1; wr_en = 1; outputFrame = crcCurrent (23 downto 16); ReadContent = 0;
STATE_WR_CRC_3
full = 0
0
shiftEnable = 0; wr_en = 0; ReadContent = 0;
1
shiftReset = 1; wr_en = 1; outputFrame = crcCurrent (15 downto 8); ReadContent = 0;
STATE_WR_CRC_4
full = 0
0
shiftEnable = 0; wr_en = 0; ReadContent = 0;
1
shiftReset = 1; wr_en = 1; outputFrame = crcCurrent (7 downto 0); ReadContent = 0;
STATE_WR_FLAG
full = 0
0
shiftEnable = 0; wr_en = 0; ReadContent = 0;
1
shiftReset = 1; wr_en = 1; outputFrame = x"7E"; ReadContent = 0;
STATE_IDLE

Figure 14.16: Control flow chart, explicit controls. CRC and frame end management in the outbound part

14.3 PPPConverter VHDL Code

```
----------------------------------------------------------------------
--  (C) COPYRIGHT Bombardier Transportation Sweden AB, 2010
--
--  We reserve all rights in this file and in the information
--  contained therein. Reproduction, use or disclosure to third
--  parties without express authority is strictly forbidden.
--
--  %name: EthIpPPP.vhd %
--  %version: 1.0 %
--  %created_by: emiglior %
--  %date_created: 02-08-2018 14:54 %
----------------------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity PPPConverter is
port (
         clk: in std_logic;
         areset_n: in std_logic;
         macClient: in std_logic_vector(47 downto 0);
         macSProcessor: in std_logic_vector(47 downto 0);

         -- interface toward/from RioFrameBuffer
         outboundReadFrameEmpty: in std_logic;
         outboundReadContentEnd: in std_logic;
         outboundReadFrame: out std_logic;
         outboundReadContent: out std_logic;
         inFrameOutbound: in std_logic_vector(7 downto 0);

         inboundWriteFrame: out std_logic;
         inboundWriteFrameAbort: out std_logic;
         inboundWriteContent: out std_logic;
         inFrameInbound: in std_logic_vector(7 downto 0);
         inboundWriteFrameFull: in std_logic;

         -- interface toward/from Xilinx buffer
         full: in std_logic;
         wr_ack: in std_logic;
```

```
          wr_en: out std_logic;
          outFrameOutbound: out std_logic_vector (7 downto 0);

          empty: in std_logic;
          rd_en: out std_logic;
          valid: in std_logic;
          outFrameInbound: out std_logic_vector (7 downto 0);

          outFlush : out std_logic;
          inFlush: out std_logic

          );
end entity;

architecture PPPModuleBehaviour of PPPConverter is

  function reversed(slv: std_logic_vector) return std_logic_vector is
    variable result: std_logic_vector(slv'reverse_range);
  begin
    for i in slv'range loop
      result(i) := slv(i);
    end loop;
    return result;
  end reversed;

component shiftMemoryOutbound is
generic (CONTENT_WIDTH : natural );
port (
          clk: in std_logic;
          areset_n: in std_logic;
          reset: in std_logic;
          shiftEnable: in std_logic;
          shiftIn: in std_logic_vector (CONTENT_WIDTH-1 downto 0);
          shiftOut: out std_logic_vector (CONTENT_WIDTH-1 downto 0)
);
end component;

component shiftMemoryInbound is
generic (CONTENT_WIDTH : natural );
port (
          clk: in std_logic;
          areset_n: in std_logic;
          reset: in std_logic;
          shiftEnable: in std_logic;
          shiftIn: in std_logic_vector (CONTENT_WIDTH-1 downto 0);
          shiftOut: out std_logic_vector (CONTENT_WIDTH-1 downto 0)
);
end component;

component Crc32Ethernet is
  port(
    d_i : in std_logic_vector(7 downto 0);
    crc_i : in std_logic_vector(31 downto 0);
    crc_o : out std_logic_vector(31 downto 0));
end component;

type stateInbound is (
                             STATE_IDLE,
                             STATE_WR_DESTINATION_1,
                             STATE_WR_DESTINATION_2,
                             STATE_WR_DESTINATION_3,
                             STATE_WR_DESTINATION_4,
                             STATE_WR_DESTINATION_5_RD_FLAG,
                             STATE_WR_DESTINATION_6_RD_ADDRESS,
                             STATE_WR_SOURCE_1_RD_CONTROL,
                             STATE_WR_SOURCE_2_RD_PROTOCOL_1,
                             STATE_WR_SOURCE_3_RD_PROTOCOL_2,
                             STATE_WR_SOURCE_4_RD_PAYLOAD,
                             STATE_WR_SOURCE_5_RD_PAYLOAD,
                             STATE_WR_SOURCE_6_RD_PAYLOAD,
                             STATE_WR_ETHTYPE_1_RD_PAYLOAD,
                             STATE_WR_ETHTYPE_2_RD_WR_PAYLOAD,
                             STATE_RD_WR_PAYLOAD,
                             STATE_WR_CRC_1,
                             STATE_WR_CRC_2,
                             STATE_WR_CRC_3,
                             STATE_WR_CRC_4,
                             STATE_END_WR_STROBE,
                             STATE_ERROR
                           );
```

```
type stateOutbound is ( STATE_IDLE,
                             STATE_UPDATE_BUFFER_CONTROL,
                             STATE_WAIT_BUFFER,
                             STATE_RD_DESTINATION_1,
                             STATE_RD_DESTINATION_2,
                             STATE_RD_DESTINATION_3,
                             STATE_RD_DESTINATION_4,
                             STATE_RD_DESTINATION_5,
                             STATE_RD_DESTINATION_6,
                             STATE_RD_SOURCE_1,
                             STATE_RD_SOURCE_2,
                             STATE_RD_SOURCE_3,
                             STATE_RD_SOURCE_4_WR_FLAG,
                             STATE_RD_SOURCE_5_WR_ADDRESS,
                             STATE_RD_SOURCE_6_WR_CONTROL,
                             STATE_RD_ETHERNET_TYPE_1_WR_PROTOCOL_1,
                             STATE_RD_ETHERNET_TYPE_2_WR_PROTOCOL_2,
                             STATE_WAIT_WR_PAYLOAD,
                             STATE_RD_WR_PAYLOAD,
                             STATE_SHIFT_LAST_OUT,
                             STATE_WR_CRC_1,
                             STATE_WR_CRC_2,
                             STATE_WR_CRC_3,
                             STATE_WR_CRC_4,
                             STATE_WR_FLAG
                           );

signal shiftResetInbound, shiftEnableInbound : std_logic;
signal inNextFrame, inChosenFrame : std_logic_vector (7 downto 0);
signal crc32EthData: std_logic_vector (7 downto 0);
signal crc32EthCurrent, crc32EthNext: std_logic_vector (31 downto 0);
signal currentStateInbound: stateInbound;
signal ethType: std_logic_vector (7 downto 0);
signal shiftResetOutbound, shiftEnableOutbound : std_logic;
signal outNextFrame, outChosenFrame : std_logic_vector (7 downto 0);
signal crc32IpData: std_logic_vector (7 downto 0);
signal crc32IpCurrent, crc32IpNext: std_logic_vector (31 downto 0);
signal currentStateOutbound: stateOutbound;
signal outputFrameOutbound, outputFrameInbound: std_logic_vector (7 downto
0);
```

```
signal inboundOutputWriteFrame: std_logic;

begin
-- inbound components
inboundShifter: shiftMemoryInbound
generic map (CONTENT_WIDTH => 8)
port map (     clk => clk, areset_n => areset_n, reset => shiftResetInbound,
shiftEnable => shiftEnableInbound,
shiftIn => inNextFrame, shiftOut => inChosenFrame);

crc32BitEthModule: Crc32Ethernet
port map ( d_i => crc32EthData, crc_i => crc32EthNext, crc_o =>
crc32EthCurrent);

-- oubound components
outboundShifter: shiftMemoryOutbound
generic map (CONTENT_WIDTH => 8)
port map (     clk => clk, areset_n => areset_n, reset =>
shiftResetOutbound, shiftEnable => shiftEnableOutbound,
shiftIn => outNextFrame, shiftOut => outChosenFrame);

crc32BitIPModule: Crc32Ethernet
port map ( d_i => crc32IpData, crc_i => crc32IpNext, crc_o =>
crc32IpCurrent);

outFrameInbound <= outputFrameInbound;
inboundWriteFrame <= inboundOutputWriteFrame;
inboundFrameCreationStateContent: process (clk, areset_n)
begin
if (areset_n='0') then
     currentStateInbound <= STATE_IDLE;
     shiftEnableInbound <= '0';
     shiftResetInbound <= '1';
     rd_en <= '0';
     inboundOutputWriteFrame <= '0';
     inboundWriteContent <= '0';
     inboundWriteFrameAbort <= '0';
     inFlush <= '1';
elsif (clk'event and clk='1') then
case currentStateInbound is

when STATE_IDLE  =>         shiftEnableInbound <= '0';
```

```
                                  shiftResetInbound <= '1';
                                  inFlush <= '0';
                                  rd_en <= '0';
                                  inboundOutputWriteFrame <= '0';
                                  inboundWriteContent <= '0';
                                  inboundWriteFrameAbort <= '0';
                                  crc32EthData <= (others => '0');
                                  crc32EthNext <= x"ffffffff";
                                  if empty = '0' then
                                  currentStateInbound <=
STATE_WR_DESTINATION_1;
                                  else currentStateInbound <= STATE_IDLE;
                                  end if;

when STATE_WR_DESTINATION_1 =>
                                  if (inboundWriteFrameFull = '0') then
                                  shiftEnableInbound <= '0';
                                  shiftResetInbound <= '0';
                                  rd_en <= '0';
                                  inboundOutputWriteFrame <= '0';
                                  inboundWriteContent <= '1';
                                  crc32EthData <= reversed (macSProcessor
(47 downto 40));
                                  crc32EthNext <= crc32EthCurrent;
                                  outputFrameInbound <= macSProcessor (47
downto 40);
                                  currentStateInbound <=
STATE_WR_DESTINATION_2;
                                  else
                                  shiftEnableInbound <= '0';
                                  shiftResetInbound <= '0';
                                  rd_en <= '0';
                                  inboundOutputWriteFrame <= '0';
                                  inboundWriteContent <= '0';
                                  currentStateInbound <=
STATE_WR_DESTINATION_1;
                                  end if;

when STATE_WR_DESTINATION_2 =>
                                  if (inboundWriteFrameFull = '0') then
                                  shiftEnableInbound <= '0';
                                  shiftResetInbound <= '0';
                                  rd_en <= '0';
                                  inboundOutputWriteFrame <= '0';
                                  inboundWriteContent <= '1';
                                  crc32EthData <= reversed(macSProcessor (39
downto 32));
                                  crc32EthNext <= crc32EthCurrent;
                                  outputFrameInbound <= macSProcessor (39
downto 32);
                                  currentStateInbound <=
STATE_WR_DESTINATION_3;
                                  else
                                  shiftEnableInbound <= '0';
                                  shiftResetInbound <= '0'
                                  rd_en <= '0';
                                  inboundOutputWriteFrame <= '0';
                                  inboundWriteContent <= '0';
                                  currentStateInbound <=
STATE_WR_DESTINATION_2;
                                  end if;

when STATE_WR_DESTINATION_3 =>
                                  if (inboundWriteFrameFull = '0') then
                                  shiftEnableInbound <= '0';
                                  shiftResetInbound <= '0';
                                  rd_en <= '0';
                                  inboundOutputWriteFrame <= '0';
                                  inboundWriteContent <= '1';
                                  crc32EthData <= reversed (macSProcessor
(31 downto 24));
                                  crc32EthNext <= crc32EthCurrent;
                                  outputFrameInbound <= macSProcessor (31
downto 24);
                                  currentStateInbound <=
STATE_WR_DESTINATION_4;
                                  else
                                  shiftEnableInbound <= '0';
                                  shiftResetInbound <= '0';
                                  rd_en <= '0';
                                  inboundOutputWriteFrame <= '0';
                                  inboundWriteContent <= '0';
                                  currentStateInbound <=
STATE_WR_DESTINATION_3;
                                  end if;
```

```
when STATE_WR_DESTINATION_4 =>
                                if (inboundWriteFrameFull = '0') then
                                shiftEnableInbound <= '0';
                                shiftResetInbound <= '0';
                                rd_en <= '1';
                                inboundOutputWriteFrame <= '0';
                                inboundWriteContent <= '1';
                                crc32EthData <= reversed( macSProcessor
(23 downto 16));
                                crc32EthNext <= crc32EthCurrent;
                                outputFrameInbound <= macSProcessor (23
downto 16);
                                currentStateInbound <=
STATE_WR_DESTINATION_5_RD_FLAG;
                                else
                                shiftEnableInbound <= '0';
                                shiftResetInbound <= '0
                                rd_en <= '1';
                                inboundOutputWriteFrame <= '0';
                                inboundWriteContent <= '0';
                                currentStateInbound <=
STATE_WR_DESTINATION_4;
                                end if;

when STATE_WR_DESTINATION_5_RD_FLAG =>
                                if (valid = '1' and inboundWriteFrameFull
= '0') then
                                shiftEnableInbound <= '0';
                                shiftResetInbound <= '0';
                                rd_en <= '1';
                                inboundOutputWriteFrame <= '0';
                                inboundWriteContent <= '1';
                                crc32EthData <= reversed(macSProcessor (15
downto 8));
                                crc32EthNext <= crc32EthCurrent;
                                outputFrameInbound <= macSProcessor (15
downto 8);
                                if inFrameInbound /= x"7E" then
currentStateInbound <= STATE_ERROR;
                                else currentStateInbound <=
STATE_WR_DESTINATION_6_RD_ADDRESS;
                                end if;
```

```
                                    else
                                    currentStateInbound <=
STATE_WR_DESTINATION_5_RD_FLAG;
                                    shiftEnableInbound <= '0';
                                    shiftResetInbound <= '0';
                                    rd_en <= '1';
                                    inboundOutputWriteFrame <= '0';
                                    inboundWriteContent <= '0';
                                    end if;

when STATE_WR_DESTINATION_6_RD_ADDRESS =>
                                    if (valid = '1' and inboundWriteFrameFull
= '0') then
                                    shiftEnableInbound <= '0';
                                    shiftResetInbound <= '0';
                                    rd_en <= '1';
                                    inboundOutputWriteFrame <= '0';
                                    inboundWriteContent <= '1';
                                    crc32EthData <= reversed (macSProcessor (7
downto 0));
                                    crc32EthNext <= crc32EthCurrent;
                                    outputFrameInbound <= macSProcessor (7
downto 0);
                                    if (inFrameInbound /= x"FF") then
currentStateInbound <= STATE_ERROR;
                                    else
                                    currentStateInbound <=
STATE_WR_SOURCE_1_RD_CONTROL;
                                    end if;
                                    else
                                    shiftEnableInbound <= '0';
                                    shiftResetInbound <= '0';
                                    rd_en <= '1';
                                    inboundOutputWriteFrame <= '0';
                                    inboundWriteContent <= '0';
                                    currentStateInbound <=
STATE_WR_DESTINATION_6_RD_ADDRESS;
                                    end if;

when STATE_WR_SOURCE_1_RD_CONTROL =>
                                    if (valid = '1' and inboundWriteFrameFull
= '0') then
                                    shiftEnableInbound <= '0';
```

```
                                 shiftResetInbound <= '0';
                                 rd_en <= '1';
                                 inboundOutputWriteFrame <= '0';
                                 inboundWriteContent <= '1';
                                 crc32EthData <= reversed (macClient (47
downto 40));
                                 crc32EthNext <= crc32EthCurrent;
                                 outputFrameInbound <= macClient (47 downto
40);
                                 if (inFrameInbound /= x"03") then
currentStateInbound <= STATE_ERROR;
                                 else
                                 currentStateInbound <=
STATE_WR_SOURCE_2_RD_PROTOCOL_1;
                                 end if;
                                 else
                                 currentStateInbound <=
STATE_WR_SOURCE_1_RD_CONTROL;
                                 shiftEnableInbound <= '0';
                                 shiftResetInbound <= '0';
                                 rd_en <= '1';
                                 inboundOutputWriteFrame <= '0';
                                 inboundWriteContent <= '0';
                                 end if;

when STATE_WR_SOURCE_2_RD_PROTOCOL_1 =>
                                 if (valid = '1' and inboundWriteFrameFull
= '0') then
                                 shiftEnableInbound <= '0';
                                 shiftResetInbound <= '0';
                                 rd_en <= '1';
                                 inboundOutputWriteFrame <= '0';
                                 inboundWriteContent <= '1';
                                 crc32EthData <= reversed (macClient (39
downto 32));
                                 crc32EthNext <= crc32EthCurrent;
                                 outputFrameInbound <= macClient (39 downto
32);
                                 if (inFrameInbound /= x"00") then
currentStateInbound <= STATE_ERROR;
                                 else
                                 currentStateInbound <=
STATE_WR_SOURCE_3_RD_PROTOCOL_2;
                                 end if;
```

```
                                else
                                shiftEnableInbound <= '0';
                                shiftResetInbound <= '0';
                                rd_en <= '1';
                                inboundOutputWriteFrame <= '0';
                                inboundWriteContent <= '0';
                                currentStateInbound <=
STATE_WR_SOURCE_2_RD_PROTOCOL_1;
                                end if;

when STATE_WR_SOURCE_3_RD_PROTOCOL_2 =>
                                if (valid = '1' and inboundWriteFrameFull
= '0') then
                                shiftEnableInbound <= '0';
                                shiftResetInbound <= '0';
                                rd_en <= '1';
                                inboundOutputWriteFrame <= '0';
                                inboundWriteContent <= '1';
                                crc32EthData <= reversed (macClient (31
downto 24));
                                crc32EthNext <= crc32EthCurrent;
                                outputFrameInbound <= macClient (31 downto
24);
                                if (inFrameInbound = x"21") then
                                ethType <= x"00";
                                currentStateInbound <=
STATE_WR_SOURCE_4_RD_PAYLOAD;
                                elsif inFrameInbound = x"81" then
                                ethType <= x"06";
                                currentStateInbound <=
STATE_WR_SOURCE_4_RD_PAYLOAD;
                                else
                                currentStateInbound <= STATE_ERROR;
                                end if;
                                else
                                shiftEnableInbound <= '0';
                                shiftResetInbound <= '0';
                                rd_en <= '1';
                                inboundOutputWriteFrame <= '0';
                                inboundWriteContent <= '0';
                                currentStateInbound <=
STATE_WR_SOURCE_3_RD_PROTOCOL_2;
                                end if;
```

```
when STATE_WR_SOURCE_4_RD_PAYLOAD =>
                                if (valid = '1' and inboundWriteFrameFull
= '0') then
                                shiftEnableInbound <= '1';
                                shiftResetInbound <= '0';
                                rd_en <= '1';
                                inboundOutputWriteFrame <= '0';
                                inboundWriteContent <= '1';
                                crc32EthData <= reversed (macClient (23
downto 16));
                                crc32EthNext <= crc32EthCurrent;
                                outputFrameInbound <= macClient (23 downto
16);
                                inNextFrame <= inFrameInbound;
                                currentStateInbound <=
STATE_WR_SOURCE_5_RD_PAYLOAD;
                                else
                                shiftEnableInbound <= '0';
                                shiftResetInbound <= '0';
                                rd_en <= '1';
                                inboundOutputWriteFrame <= '0';
                                inboundWriteContent <= '0';
                                currentStateInbound <=
STATE_WR_SOURCE_4_RD_PAYLOAD;
                                end if;

when STATE_WR_SOURCE_5_RD_PAYLOAD =>
                                if (valid = '1' and inboundWriteFrameFull
= '0') then
                                shiftEnableInbound <= '1';
                                shiftResetInbound <= '0';
                                rd_en <= '1';
                                inboundOutputWriteFrame <= '0';
                                inboundWriteContent <= '1';
                                crc32EthData <= reversed (macClient (15
downto 8));
                                crc32EthNext <= crc32EthCurrent;
                                inNextFrame <= inFrameInbound;
                                outputFrameInbound <= macClient (15 downto
8);
                                currentStateInbound <=
STATE_WR_SOURCE_6_RD_PAYLOAD;
```

```
                                  else
                                  shiftEnableInbound <= '0';
                                  shiftResetInbound <= '0';
                                  rd_en <= '1';
                                  inboundOutputWriteFrame <= '0';
                                  inboundWriteContent <= '0';
                                  currentStateInbound <=
STATE_WR_SOURCE_5_RD_PAYLOAD;
                                  end if;

when STATE_WR_SOURCE_6_RD_PAYLOAD =>
                                  if (valid = '1' and inboundWriteFrameFull
= '0') then
                                  shiftEnableInbound <= '1';
                                  shiftResetInbound <= '0';
                                  rd_en <= '1';
                                  inboundOutputWriteFrame <= '0';
                                  inboundWriteContent <= '1';
                                  crc32EthData <= reversed (macClient (7
downto 0));
                                  crc32EthNext <= crc32EthCurrent;
                                  inNextFrame <= inFrameInbound;
                                  outputFrameInbound <= macClient (7 downto
0);
                                  currentStateInbound <=
STATE_WR_ETHTYPE_1_RD_PAYLOAD;
                                  else
                                  shiftEnableInbound <= '0';
                                  shiftResetInbound <= '0';
                                  rd_en <= '1';
                                  inboundOutputWriteFrame <= '0';
                                  inboundWriteContent <= '0';
                                  currentStateInbound <=
STATE_WR_SOURCE_6_RD_PAYLOAD;
                                  end if;

when STATE_WR_ETHTYPE_1_RD_PAYLOAD =>
                                  if (valid = '1' and inboundWriteFrameFull
= '0') then
                                  shiftEnableInbound <= '1';
                                  shiftResetInbound <= '0';
                                  rd_en <= '1';
                                  inboundOutputWriteFrame <= '0';
                                  inboundWriteContent <= '1';
                                  crc32EthData <= reversed (x"08");
                                  crc32EthNext <= crc32EthCurrent;
                                  inNextFrame <= inFrameInbound;
                                  outputFrameInbound <= x"08";
                                  currentStateInbound <=
STATE_WR_ETHTYPE_2_RD_WR_PAYLOAD;
                                  else
                                  shiftEnableInbound <= '0';
                                  shiftResetInbound <= '0';
                                  rd_en <= '1';
                                  inboundOutputWriteFrame <= '0';
                                  inboundWriteContent <= '0';
                                  currentStateInbound <=
STATE_WR_ETHTYPE_1_RD_PAYLOAD;
                                  end if;

when STATE_WR_ETHTYPE_2_RD_WR_PAYLOAD =>
                                  if (valid = '1' and inboundWriteFrameFull
= '0') then
                                  shiftEnableInbound <= '1';
                                  shiftResetInbound <= '0';
                                  inNextFrame <= inFrameInbound;
                                  rd_en <= '1';
                                  inboundOutputWriteFrame <= '0';
                                  inboundWriteContent <= '1';
                                  crc32EthData <= reversed (ethType);
                                  crc32EthNext <= crc32EthCurrent;
                                  outputFrameInbound <= ethType;
                                  currentStateInbound <=
STATE_RD_WR_PAYLOAD;
                                  else
                                  shiftEnableInbound <= '0';
                                  shiftResetInbound <= '0';
                                  rd_en <= '1';
                                  inboundOutputWriteFrame <= '0';
                                  inboundWriteContent <= '0';
                                  currentStateInbound <=
STATE_WR_ETHTYPE_2_RD_WR_PAYLOAD;
                                  end if;
```

```
when STATE_RD_WR_PAYLOAD =>  if (valid = '1' and inboundWriteFrameFull =
'0') then
                                 shiftEnableInbound <= '1';
                                 shiftResetInbound <= '0';
                                 rd_en <= '1';
                                 inboundOutputWriteFrame <= '0';
                                 inboundWriteContent <= '1';
                                 outputFrameInbound <= inChosenFrame;
                                 crc32EthData <= reversed (inFrameInbound);
                                 crc32EthNext <= crc32EthCurrent;
                                 inNextFrame <= inFrameInbound;
                                 if inFrameInbound = x"7E" then
currentStateInbound <= STATE_WR_CRC_1;--STATE_SHIFT_LAST_OUT;
                                 else currentStateInbound <=
STATE_RD_WR_PAYLOAD;
                                 end if;
                                 else
                                 shiftEnableInbound <= '0';
                                 shiftResetInbound <= '0';
                                 rd_en <= '1';
                                 inboundOutputWriteFrame <= '0';
                                 inboundWriteContent <= '0';
                                 currentStateInbound <=
STATE_RD_WR_PAYLOAD;
                                 end if;

when STATE_WR_CRC_1 =>       if (inboundWriteFrameFull = '0') then
                                 shiftEnableInbound <= '0';
                                 shiftResetInbound <= '0';
                                 rd_en <= '0';
                                 inboundOutputWriteFrame <= '0';
                                 inboundWriteContent <= '1';
                                 outputFrameInbound <= crc32EthCurrent (31
downto 24);
                                 currentStateInbound <= STATE_WR_CRC_2;
                                 else
                                 shiftEnableInbound <= '0';
                                 shiftResetInbound <= '0';
                                 rd_en <= '0';
                                 inboundOutputWriteFrame <= '0';
                                 inboundWriteContent <= '0';
                                 currentStateInbound <= STATE_WR_CRC_1;
```

```
                                        end if;

when STATE_WR_CRC_2 =>          if (inboundWriteFrameFull = '0') then
                                        shiftEnableInbound <= '0';
                                        shiftResetInbound <= '0';
                                        rd_en <= '0';
                                        inboundOutputWriteFrame <= '0';
                                        inboundWriteContent <= '1';
                                        outputFrameInbound <= crc32EthCurrent (23
downto 16);
                                        currentStateInbound <= STATE_WR_CRC_3;
                                        else
                                        shiftEnableInbound <= '0';
                                        shiftResetInbound <= '0';
                                        rd_en <= '0';
                                        inboundOutputWriteFrame <= '0';
                                        inboundWriteContent <= '0';
                                        currentStateInbound <= STATE_WR_CRC_2;
                                        end if;

when STATE_WR_CRC_3 =>          if (inboundWriteFrameFull = '0') then
                                        shiftEnableInbound <= '0';
                                        shiftResetInbound <= '0';
                                        rd_en <= '0';
                                        inboundOutputWriteFrame <= '0';
                                        inboundWriteContent <= '1';
                                        outputFrameInbound <= crc32EthCurrent (15
downto 8);
                                        currentStateInbound <= STATE_WR_CRC_4;
                                        else
                                        shiftEnableInbound <= '0';
                                        shiftResetInbound <= '0';
                                        rd_en <= '0';
                                        inboundOutputWriteFrame <= '0';
                                        inboundWriteContent <= '0';
                                        currentStateInbound <= STATE_WR_CRC_3;
                                        end if;


when STATE_WR_CRC_4 =>          if (inboundWriteFrameFull = '0') then
                                        shiftEnableInbound <= '0';
```

```
                                 shiftResetInbound <= '0';
                                 rd_en <= '0';
                                 inboundOutputWriteFrame <= '0';
                                 inboundWriteContent <= '1';
                                 outputFrameInbound <= crc32EthCurrent (7
downto 0);
                                 currentStateInbound <=
STATE_END_WR_STROBE;
                                 else
                                 shiftEnableInbound <= '0';
                                 shiftResetInbound <= '0';
                                 rd_en <= '0';
                                 inboundOutputWriteFrame <= '0';
                                 inboundWriteContent <= '0';
                                 currentStateInbound <= STATE_WR_CRC_4;
                                 end if;

when STATE_END_WR_STROBE =>  shiftEnableInbound <= '0';
                                 shiftResetInbound <= '1';
                                 rd_en <= '0';
                                 inboundOutputWriteFrame <= '1';
                                 inboundWriteContent <= '0';
                                 outputFrameInbound <= inChosenFrame; -- in
this way it is reset to zero
                                 currentStateInbound <= STATE_IDLE;

when STATE_ERROR =>          shiftEnableInbound <= '0';
                                 shiftResetInbound <= '1';
                                 rd_en <= '0';
                                 inboundOutputWriteFrame <= '0';
                                 inboundWriteContent <= '0';
                                 inboundWriteFrameAbort <= '1';
                                 shiftResetInbound <= '0';
                                 currentStateInbound <= STATE_IDLE;

when others =>     currentStateInbound <= STATE_ERROR;

end case;
end if;
end process;
```

```
-- outbound process

outFrameOutbound <= outputFrameOutbound;
outboundFrameCreationStateContent: process (clk, areset_n)
begin
     if (areset_n='0') then
          shiftEnableOutbound <= '0';
          shiftResetOutbound <= '1';
          outputFrameOutbound <= (others => '0');
          wr_en <= '0';
          outFlush <= '1';
          outboundReadFrame <= '0';
          currentStateOutbound <= STATE_IDLE;
          outboundReadContent <= '0';

     elsif (clk'event and clk='1') then

case currentStateOutbound is

when STATE_IDLE =>
                                  shiftEnableOutbound <= '0';
                                  shiftResetOutbound <= '1';
                                  outFlush <= '0';
                                  wr_en <= '0';
                                  outboundReadFrame <= '0';
                                  outboundReadContent <= '0';
                                  crc32IpData <= (others => '0');
                                  if outboundReadFrameEmpty = '0' then
                                  currentStateOutbound <=
STATE_UPDATE_BUFFER_CONTROL;
                                  else currentStateOutbound <= STATE_IDLE;
                                  end if;

when STATE_UPDATE_BUFFER_CONTROL =>
                                  shiftEnableOutbound <= '0';
                                  shiftResetOutbound <= '0';
                                  wr_en <= '0';
                                  outboundReadFrame <= '0';
                                  outboundReadContent <= '1';
                                  currentStateOutbound <= STATE_WAIT_BUFFER;
```

```
when STATE_WAIT_BUFFER =>
                                  shiftEnableOutbound <= '0';
                                  shiftResetOutbound <= '0';
                                  wr_en <= '0';
                                  outboundReadFrame <= '0';
                                  outboundReadContent <= '1';
                                  currentStateOutbound <=
STATE_RD_DESTINATION_1;

when STATE_RD_DESTINATION_1 =>
                                  shiftEnableOutbound <= '0';
                                  shiftResetOutbound <= '0';
                                  wr_en <= '0';
                                  outboundReadFrame <= '0';
                                  outboundReadContent <= '1';
                                  currentStateOutbound <=
STATE_RD_DESTINATION_2;

when STATE_RD_DESTINATION_2 =>
                                  shiftEnableOutbound <= '0';
                                  shiftResetOutbound <= '0';
                                  wr_en <= '0';
                                  outboundReadFrame <= '0';
                                  outboundReadContent <= '1';
                                  currentStateOutbound <=
STATE_RD_DESTINATION_3;

when STATE_RD_DESTINATION_3 =>
                                  shiftEnableOutbound <= '0';
                                  shiftResetOutbound <= '0';
                                  wr_en <= '0';
                                  outboundReadFrame <= '0';
                                  outboundReadContent <= '1';
                                  currentStateOutbound <=
STATE_RD_DESTINATION_4;

when STATE_RD_DESTINATION_4 =>
                                  shiftEnableOutbound <= '0';
                                  shiftResetOutbound <= '0';
                                  wr_en <= '0';
                                  outboundReadFrame <= '0';
                                  outboundReadContent <= '1';
```

```
                                 currentStateOutbound <=
STATE_RD_DESTINATION_5;

when STATE_RD_DESTINATION_5 =>
                                 shiftEnableOutbound <= '0';
                                 shiftResetOutbound <= '0';
                                 wr_en <= '0';
                                 outboundReadFrame <= '0';
                                 outboundReadContent <= '1';
                                 currentStateOutbound <=
STATE_RD_DESTINATION_6;

when STATE_RD_DESTINATION_6 =>
                                 shiftEnableOutbound <= '0';
                                 shiftResetOutbound <= '0';
                                 wr_en <= '0';
                                 outboundReadFrame <= '0';
                                 outboundReadContent <= '1';
                                 currentStateOutbound <= STATE_RD_SOURCE_1;

when STATE_RD_SOURCE_1 =>   shiftEnableOutbound <= '0';
                                 shiftResetOutbound <= '0';
                                 wr_en <= '0';
                                 outboundReadFrame <= '0';
                                 outboundReadContent <= '1';
                                 currentStateOutbound <= STATE_RD_SOURCE_2;

when STATE_RD_SOURCE_2 =>   shiftEnableOutbound <= '0';
                                 shiftResetOutbound <= '0';
                                 wr_en <= '0';
                                 outboundReadFrame <= '0';
                                 outboundReadContent <= '1';
                                 currentStateOutbound <= STATE_RD_SOURCE_3;

when STATE_RD_SOURCE_3 =>   shiftEnableOutbound <= '0';
                                 shiftResetOutbound <= '0';
                                 wr_en <= '0';
                                 outboundReadFrame <= '0';
                                 outboundReadContent <= '1';
                                 currentStateOutbound <=
STATE_RD_SOURCE_4_WR_FLAG;
```

```
when STATE_RD_SOURCE_4_WR_FLAG =>
                                 shiftEnableOutbound <= '1';
                                 shiftResetOutbound <= '0';
                                 outNextFrame <= x"7E";
                                 wr_en <= '0';
                                 crc32IpNext <= x"ffffffff";
                                 outboundReadFrame <= '0';
                                 outboundReadContent <= '1';
                                 currentStateOutbound <=
STATE_RD_SOURCE_5_WR_ADDRESS;

when STATE_RD_SOURCE_5_WR_ADDRESS =>
                                 shiftEnableOutbound <= '1';
                                 outNextFrame <= x"FF";
                                 shiftResetOutbound <= '0';
                                 wr_en <= '0';
                                 outboundReadFrame <= '0';
                                 outboundReadContent <= '1';
                                 currentStateOutbound <=
STATE_RD_SOURCE_6_WR_CONTROL;

when STATE_RD_SOURCE_6_WR_CONTROL =>
                                 shiftEnableOutbound <= '1';
                                 outNextFrame <= x"03";
                                 wr_en <= '0';
                                 outboundReadFrame <= '0';
                                 outboundReadContent <= '1';
                                 currentStateOutbound <=
STATE_RD_ETHERNET_TYPE_1_WR_PROTOCOL_1;

when STATE_RD_ETHERNET_TYPE_1_WR_PROTOCOL_1  =>
                                 shiftEnableOutbound <= '1';
                                 outNextFrame <= x"00";
                                 wr_en <= '0';
                                 outboundReadFrame <= '0';
                                 outboundReadContent <= '1';
                                 currentStateOutbound <=
STATE_RD_ETHERNET_TYPE_2_WR_PROTOCOL_2;

when STATE_RD_ETHERNET_TYPE_2_WR_PROTOCOL_2 =>
                                 shiftEnableOutbound <= '1';
                                 if inFrameOutbound = x"00" then
                                 outNextFrame <= x"21";
                                 elsif inFrameOutbound = x"06" then
                                 outNextFrame <= x"81";
                                 else currentStateOutbound <= STATE_IDLE; -
-error in the protocol
                                 shiftResetOutbound <= '1';
                                 shiftEnableOutbound <= '0';
                                 outboundReadFrame <= '1';
                                 outboundReadContent <= '0';
                                 end if;
                                 wr_en <= '0';
                                 crc32IpData <= reversed (outChosenFrame);
                                 crc32IpNext <= crc32IpCurrent;
                                 outboundReadFrame <= '0';
                                 outboundReadContent <= '1';
                                 currentStateOutbound <=
STATE_WAIT_WR_PAYLOAD;

when STATE_WAIT_WR_PAYLOAD =>
                                 if (full = '0') then
                                 shiftEnableOutbound <= '1';
                                 outNextFrame <= inFrameOutbound;
                                 wr_en <= '0';
                                 crc32IpData <= reversed (outChosenFrame);
                                 crc32IpNext <= crc32IpCurrent;
                                 outboundReadFrame <= '0';
                                 outboundReadContent <= '1';
                                 currentStateOutbound <=
STATE_RD_WR_PAYLOAD;
                                 else
                                 shiftEnableOutbound <= '0';
                                 wr_en <= '0';
                                 outboundReadContent <= '0';
                                 currentStateOutbound <=
STATE_WAIT_WR_PAYLOAD;
                                 end if;

when STATE_RD_WR_PAYLOAD =>
                                 if (full = '0') then
                                 shiftEnableOutbound <= '1';
                                 outNextFrame <= inFrameOutbound;
```

```
                                wr_en <= '1';
                                crc32IpData <= reversed (outChosenFrame);
                                crc32IpNext <= crc32IpCurrent;
                                outputFrameOutbound <= outChosenFrame;
                                outboundReadFrame <= '0';
                                outboundReadContent <= '1';
                                if outboundReadContentEnd = '1' then
                                currentStateOutbound <=
STATE_SHIFT_LAST_OUT;
                                else currentStateOutbound <=
STATE_RD_WR_PAYLOAD;
                                end if;
                                else
                                shiftEnableOutbound <= '0';
                                wr_en <= '0';
                                outboundReadContent <= '0';
                                currentStateOutbound <=
STATE_RD_WR_PAYLOAD;
                                end if;

when STATE_SHIFT_LAST_OUT =>
                                if (full = '0') then
                                shiftEnableOutbound <= '1';
                                wr_en <= '1';
                                crc32IpData <= reversed (outChosenFrame);
                                crc32IpNext <= crc32IpCurrent;
                                outputFrameOutbound <= outChosenFrame;
                                outboundReadFrame <= '0';
                                outboundReadContent <= '0';
                                currentStateOutbound<= STATE_WR_CRC_1;
                                else
                                shiftEnableOutbound <= '0';
                                wr_en <= '0';
                                outboundReadContent <= '0';
                                currentStateOutbound <=
STATE_SHIFT_LAST_OUT;
                                end if;

when STATE_WR_CRC_1 =>
                                if (full = '0') then
                                shiftEnableOutbound <= '0';
                                shiftResetOutbound <= '1';
```

```
                                  wr_en <= '1';
                                  outputFrameOutbound <= crc32IpCurrent (31
downto 24);
                                  outboundReadFrame <= '1';
                                  outboundReadContent <= '0';
                                  currentStateOutbound <= STATE_WR_CRC_2;
                                  else
                                  shiftEnableOutbound <= '0';
                                  wr_en <= '0';
                                  outboundReadContent <= '0';
                                  currentStateOutbound <= STATE_WR_CRC_1;
                                  end if;

when STATE_WR_CRC_2 =>
                                  if (full = '0') then
                                  shiftEnableOutbound <= '0';
                                  shiftResetOutbound <= '1';
                                  wr_en <= '1';
                                  outboundReadFrame <= '0';
                                  outboundReadContent <= '0';
                                  outputFrameOutbound <= crc32IpCurrent (23
downto 16);
                                  currentStateOutbound <= STATE_WR_CRC_3;
                                  else
                                  shiftEnableOutbound <= '0';
                                  wr_en <= '0';
                                  outboundReadContent <= '0';
                                  currentStateOutbound <= STATE_WR_CRC_2;
                                  end if;

when STATE_WR_CRC_3 =>
                                  if (full = '0') then
                                  shiftEnableOutbound <= '0';
                                  shiftResetOutbound <= '1';
                                  wr_en <= '1';
                                  outputFrameOutbound <= crc32IpCurrent (15
downto 8);
                                  outboundReadFrame <= '0';
                                  outboundReadContent <= '0';
                                  currentStateOutbound <= STATE_WR_CRC_4;
                                  else
```

```
                                   shiftEnableOutbound <= '0';
                                   wr_en <= '0';
                                   outboundReadContent <= '0';
                                   currentStateOutbound <= STATE_WR_CRC_3;
                                   end if;

when STATE_WR_CRC_4 =>
                                   if (full = '0') then
                                   shiftEnableOutbound <= '0';
                                   shiftResetOutbound <= '1';
                                   wr_en <= '1';
                                   outboundReadFrame <= '0';
                                   outboundReadContent <= '0';
                                   outputFrameOutbound <= crc32IpCurrent (7
downto 0);
                                   currentStateOutbound <= STATE_WR_FLAG;
                                   else
                                   shiftEnableOutbound <= '0';
                                   wr_en <= '0';
                                   outboundReadContent <= '0';
                                   currentStateOutbound <= STATE_WR_CRC_4;
                                   end if;

when STATE_WR_FLAG =>
                                   if (full = '0') then
                                   shiftEnableOutbound <= '0';
                                   shiftResetOutbound <= '1';
                                   wr_en <= '1';
                                   outboundReadFrame <= '0';
                                   outboundReadContent <= '0';
                                   outputFrameOutbound <= x"7E";
                                   currentStateOutbound <= STATE_IDLE;
                                   else
                                   shiftEnableOutbound <= '0';
                                   wr_en <= '0';
                                   outboundReadContent <= '0';
                                   currentStateOutbound <= STATE_WR_FLAG;
                                   end if;

when others     => currentStateOutbound <= STATE_IDLE;
```

```
end case;
end if;
end process;
end architecture;

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity shiftMemoryOutbound is
generic (CONTENT_WIDTH : natural );
port (
         clk: in std_logic;
         areset_n: in std_logic;
         reset: in std_logic;
         shiftEnable: in std_logic;
         shiftIn: in std_logic_vector (CONTENT_WIDTH-1 downto 0);
         shiftOut: out std_logic_vector (CONTENT_WIDTH-1 downto 0)
);
end entity;

architecture behaviourShiftMemory of shiftMemoryOutbound is

signal slot1, slot2, slot3, slot4: std_logic_vector (CONTENT_WIDTH-1 downto
0);
begin

shifting: process (areset_n, shiftEnable, reset, clk) --clk
begin

if (areset_n = '0' or reset = '1') then slot1 <= x"7e";
                                              slot2 <= x"7e";
                                              slot3 <= x"7e";
                                              slot4 <= x"7e";
                                              shiftOut <= x"7e";
elsif (clk ' event and clk='1') then
if (shiftEnable= '1') then  slot1 <= shiftIn;
                                 slot2 <= slot1;
                                 slot3 <= slot2;
                                 slot4 <= slot3;
                                 shiftOut <= slot4;
```

```
end if;
end if;
end process;
end architecture;

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity shiftMemoryInbound is
generic (CONTENT_WIDTH : natural );
port (
          clk: in std_logic;
          areset_n: in std_logic;
          reset: in std_logic;
          shiftEnable: in std_logic;
          shiftIn: in std_logic_vector (CONTENT_WIDTH-1 downto 0);
          shiftOut: out std_logic_vector (CONTENT_WIDTH-1 downto 0)
);
end entity;

architecture behaviourShiftMemory of shiftMemoryInbound is

signal slot1, slot2, slot3, slot4: std_logic_vector (CONTENT_WIDTH-1 downto
0);
begin

shifting: process (areset_n, shiftEnable, reset, clk) --clk
begin

if (areset_n = '0' or reset = '1') then slot1 <= x"00";
                                             slot2 <= x"00";
                                             slot3 <= x"00";
                                             slot4 <= x"00";
                                             shiftOut <= x"00";
elsif (clk ' event and clk='1') then
if (shiftEnable= '1') then  slot1 <= shiftIn;
                                      slot2 <= slot1;
                                      slot3 <= slot2;
                                      slot4 <= slot3;
                                      shiftOut <= slot4;
```

```
end if;
end if;
end process;
end architecture;
```

14.4 Testbench VHDL codes

In this section, all the VHDL codes used for the testbenches are reported.

14.4.1 ShiftMemory testbench

```
----------------------------------------------------------------
--  (C) COPYRIGHT Bombardier Transportation Sweden AB, 2014
--  We reserve all rights in this file and in the information
--  contained therein. Reproduction, use or disclosure to third
--  parties without express authority is strictly forbidden.
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity tb_shiftmem is
end entity;

architecture test_behaviour of tb_shiftmem is

component shiftMemoryOutbound is
generic (CONTENT_WIDTH : natural );
port (
         clk: in std_logic;
         areset_n: in std_logic;
         reset: in std_logic;
         shiftEnable: in std_logic;
         shiftIn: in std_logic_vector (CONTENT_WIDTH-1 downto 0);
         shiftOut: out std_logic_vector (CONTENT_WIDTH-1 downto 0)
);
end component;

component shiftMemoryInbound is
generic (CONTENT_WIDTH : natural );
port (
         clk: in std_logic;
         areset_n: in std_logic;
         reset: in std_logic;
         shiftEnable: in std_logic;
```

```
          shiftIn: in std_logic_vector (CONTENT_WIDTH-1 downto 0);
          shiftOut: out std_logic_vector (CONTENT_WIDTH-1 downto 0)
);
end component;

signal clk : std_logic;
signal areset_n: std_logic := '0';
signal resetIn, resetOut, enableIn, enableOut : std_logic;
signal inFrameInbound, inFrameOutbound, outFrameInbound, outFrameOutbound:
std_logic_vector ( 7 downto 0);

begin

UUTOutbound: shiftMemoryOutbound
generic map (CONTENT_WIDTH => 8 )
port map (
          clk => clk,
          areset_n => areset_n,
          reset => resetOut,
          shiftEnable => enableOut,
          shiftIn => inFrameOutbound,
          shiftOut => outFrameOutbound
);

UUTInbound: shiftMemoryInbound
generic map (CONTENT_WIDTH => 8 )
port map (
          clk => clk,
          areset_n => areset_n,
          reset => resetIn,
          shiftEnable => enableIn,
          shiftIn => inFrameInbound,
          shiftOut => outFrameInbound
);
inboundstimuli: process
begin
wait for 80 ns;
resetIn <= '0';
inFrameInbound <= x"01";
enableIn <= '1';
wait for 40 ns;
```

```
inFrameInbound <= x"02";
enableIn <= '1';
wait for 40 ns;
inFrameInbound <= x"03";
enableIn <= '1';
wait for 40 ns;
inFrameInbound <= x"04";
enableIn <= '1';
wait for 40 ns;
inFrameInbound <= x"05";
enableIn <= '1';
wait for 40 ns;
inFrameInbound <= x"06";
enableIn <= '1';
wait for 40 ns;
enableIn <= '0';
wait for 40 ns;
resetIn <= '1';
end process;

outboundstimuli: process
begin
wait for 80 ns;
resetOut <= '0';
inFrameOutbound <= x"01";
enableOut <= '1';
wait for 40 ns;
inFrameOutbound <= x"02";
enableOut <= '1';
wait for 40 ns;
inFrameOutbound <= x"03";
enableOut <= '1';
wait for 40 ns;
inFrameOutbound <= x"04";
enableOut <= '1';
wait for 40 ns;
inFrameOutbound <= x"05";
enableOut <= '1';
wait for 40 ns;
inFrameOutbound <= x"06";
```

```
enableOut <= '1';
wait for 40 ns;
enableOut <= '0';
wait for 40 ns;
resetOut <= '1';
end process;

  ClockGenerator: process
  begin
    clk <= '1';
    wait for 20 ns;
    clk <= '0';
    wait for 20 ns;
  end process;

-- Asynchronous reset generation
areset_n <= '1' after 15 ns;

end architecture;
```

14.4.2 PPPConverter inbound and outbound testbench

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity testbenchPPPBuffers is
end entity;

architecture testbenchBehaviour of testbenchPPPBuffers is

component PPPConverter is
port (
         clk: in std_logic;
         areset_n: in std_logic;
         macClient: in std_logic_vector(47 downto 0);
         macSProcessor: in std_logic_vector(47 downto 0);
         -- interface toward/from RioFrameBuffer
         outboundReadFrameEmpty: in std_logic;
         outboundReadContentEnd: in std_logic;
         outboundReadFrame: out std_logic;
```

```
          outboundReadContent: out std_logic;
          inFrameOutbound: in std_logic_vector(7 downto 0);
          inboundWriteFrame: out std_logic;
          inboundWriteFrameAbort: out std_logic;
          inboundWriteContent: out std_logic;
          inFrameInbound: in std_logic_vector(7 downto 0);
          inboundWriteFrameFull: in std_logic;
          -- interface toward/from Xilinx buffer
          full: in std_logic;
          wr_ack: in std_logic;
          wr_en: out std_logic;
          outFrameOutbound: out std_logic_vector (7 downto 0);
          empty: in std_logic;
          rd_en: out std_logic;
          valid: in std_logic;
          outFrameInbound: out std_logic_vector (7 downto 0);
          outFlush : out std_logic;
          inFlush: out std_logic
          );
end component;

component FrameBuffer is
  generic(
    SIZE_ADDRESS_WIDTH : natural := 6;
    CONTENT_ADDRESS_WIDTH : natural := 8;
    CONTENT_WIDTH : natural := 32;
    MAX_PACKET_SIZE : natural := 69);
  port(
    clk : in std_logic;
    areset_n : in std_logic;
    inboundWriteFrameFull_o : out std_logic;
    inboundWriteFrame_i : in std_logic;
    inboundWriteFrameAbort_i : in std_logic;
    inboundWriteContent_i : in std_logic;
    inboundWriteContentData_i : in std_logic_vector(CONTENT_WIDTH-1 downto 
0);
    inboundReadFrameEmpty_o : out std_logic;
    inboundReadFrame_i : in std_logic;
    inboundReadFrameRestart_i : in std_logic;
    inboundReadFrameAborted_o : out std_logic;
    inboundReadFrameSize_o : out std_logic_vector(CONTENT_ADDRESS_WIDTH-1
downto 0);
    inboundReadContentEmpty_o : out std_logic;
    inboundReadContent_i : in std_logic;
    inboundReadContentEnd_o : out std_logic;
    inboundReadContentData_o : out std_logic_vector(CONTENT_WIDTH-1 downto
0);
    outboundWriteFrameFull_o : out std_logic;
    outboundWriteFrame_i : in std_logic;
    outboundWriteFrameAbort_i : in std_logic;
    outboundWriteContent_i : in std_logic;
    outboundWriteContentData_i : in std_logic_vector(CONTENT_WIDTH-1 downto
0);
    outboundReadFrameEmpty_o : out std_logic;
    outboundReadFrame_i : in std_logic;
    outboundReadFrameRestart_i : in std_logic;
    outboundReadFrameAborted_o : out std_logic;
    outboundReadFrameSize_o : out std_logic_vector(CONTENT_ADDRESS_WIDTH-1
downto 0);
    outboundReadContentEmpty_o : out std_logic;
    outboundReadContent_i : in std_logic;
    outboundReadContentEnd_o : out std_logic;
    outboundReadContentData_o : out std_logic_vector(CONTENT_WIDTH-1 downto
0));
end component;

COMPONENT FIFO_BUFFER
  PORT (
    rst : IN STD_LOGIC;
    wr_clk : IN STD_LOGIC;
    rd_clk : IN STD_LOGIC;
    din : IN STD_LOGIC_VECTOR(7 DOWNTO 0);
    wr_en : IN STD_LOGIC;
    rd_en : IN STD_LOGIC;
    dout : OUT STD_LOGIC_VECTOR(7 DOWNTO 0);
    full : OUT STD_LOGIC;
    wr_ack : OUT STD_LOGIC;
    empty : OUT STD_LOGIC;
    valid : OUT STD_LOGIC;
    rd_data_count : OUT STD_LOGIC_VECTOR(7 DOWNTO 0);
    wr_data_count : OUT STD_LOGIC_VECTOR(7 DOWNTO 0)
  );
```

```
END COMPONENT;


signal clk: std_logic;
signal areset_n: std_logic:='0';
signal macClient: std_logic_vector (47 downto 0):= x"111213141516";
signal macSProcessor: std_logic_vector (47 downto 0):= x"212223242526";

constant CONTENT_WIDTH: natural := 8;
constant CONTENT_ADDRESS_WIDTH: natural := 14;
constant SIZE_ADDRESS_WIDTH: natural := 6;      -- 2**6  =  64 frames (max)
constant MAX_PACKET_SIZE: natural := 1520;
-- interface toward/from RioFrameBuffer
signal outboundReadFrameEmpty:  std_logic;
signal outboundReadContentEnd:  std_logic;
signal outboundReadFrame:  std_logic;
signal outboundReadContent:  std_logic;
signal inFrameOutbound: std_logic_vector(7 downto 0); -- output of the
riobuffer --> contentdata
signal outboundWriteContent: std_logic := '0';
signal outboundWriteFrame: std_logic :='0';
signal inboundWriteFrame:  std_logic;
signal inboundWriteFrameAbort:  std_logic;
signal inboundWriteContent:  std_logic;
signal inFrameInbound:  std_logic_vector(7 downto 0); -- one input of the
test
signal inboundWriteFrameFull:  std_logic;
-- interface toward/from Xilinx buffer
signal full:  std_logic := '0';
signal wr_ack:  std_logic;
signal wr_en, wr_enIN:  std_logic;-- output I want to read
signal outFrameOutbound:  std_logic_vector (7 downto 0); -- output I want
to read
signal empty:  std_logic := '1';
signal rd_en:  std_logic;
signal outFrameInbound:  std_logic_vector (7 downto 0); -- input of the
buffer --> inboundWriteContentData_i
signal inFlush, outFlush: std_logic;


-- testbench signals
signal opensignal: std_logic :='0';
signal openvectorcontent: std_logic_vector (CONTENT_WIDTH-1 downto 0) :=
(others => '0');
```

```
signal openvectoraddress: std_logic_vector (CONTENT_ADDRESS_WIDTH-1 downto
0):= (others => '0');
signal outbIN, outbOUT: std_logic_vector (7 downto 0);
signal inbOUT, inbIN: std_logic_vector (7 downto 0);
signal inboundReadContent, inboundReadFrame, rd_enOut: std_logic:='0';
signal valid, validIN, emptyIN, fullIN, wr_ackIN: std_logic;
begin
UUT: PPPConverter
port map (
              clk => clk,
              areset_n => areset_n,
              macClient => macClient,
              macSProcessor => macSProcessor,
              -- interface toward/from RioFrameBuffer
              outboundReadFrameEmpty => outboundReadFrameEmpty,
              outboundReadContentEnd => outboundReadContentEnd,
              outboundReadFrame => outboundReadFrame,
              outboundReadContent => outboundReadContent,
              inFrameOutbound => inFrameOutbound,
              inboundWriteFrame => inboundWriteFrame,
              inboundWriteFrameAbort => inboundWriteFrameAbort,
              inboundWriteContent => inboundWriteContent,
              inFrameInbound => inFrameInbound,
              inboundWriteFrameFull => inboundWriteFrameFull,
              -- interface toward/from Xilinx buffer
              full => full,
              wr_ack => wr_ack,
              wr_en => wr_en,
              outFrameOutbound => outFrameOutbound,
              empty => empty,
              rd_en => rd_en,
              valid => valid,
              outFrameInbound => outFrameInbound
          );

ethbuffer: FrameBuffer
  generic map (
    SIZE_ADDRESS_WIDTH => SIZE_ADDRESS_WIDTH,
    CONTENT_ADDRESS_WIDTH => CONTENT_ADDRESS_WIDTH,
    CONTENT_WIDTH => CONTENT_WIDTH,
    MAX_PACKET_SIZE => MAX_PACKET_SIZE )
```

```
 port map (
             clk => clk,
             areset_n => areset_n,
             inboundWriteFrameFull_o => inboundWriteFrameFull,
             inboundWriteFrame_i => inboundWriteFrame,
             inboundWriteFrameAbort_i => inboundWriteFrameAbort,
             inboundWriteContent_i => inboundWriteContent,
             inboundWriteContentData_i  => outFrameInbound,
             inboundReadFrameEmpty_o => opensignal,
             inboundReadFrame_i => inboundReadFrame,
             inboundReadFrameRestart_i => opensignal,
             inboundReadFrameAborted_o => opensignal,
             inboundReadFrameSize_o => openvectoraddress,
             inboundReadContentEmpty_o => opensignal,
             inboundReadContent_i => inboundReadContent,
             inboundReadContentEnd_o => opensignal,
             inboundReadContentData_o => inbOUT,
             outboundWriteFrameFull_o => opensignal,
             outboundWriteFrame_i => outboundWriteFrame,
             outboundWriteFrameAbort_i => opensignal,
             outboundWriteContent_i => outboundWriteContent,
             outboundWriteContentData_i => outbIN,
             outboundReadFrameEmpty_o => outboundReadFrameEmpty,
             outboundReadFrame_i => outboundReadFrame,
             outboundReadFrameRestart_i => opensignal,
             outboundReadFrameAborted_o => opensignal,
             outboundReadFrameSize_o => openvectoraddress,
             outboundReadContentEmpty_o => opensignal,
             outboundReadContent_i => outboundReadContent,
             outboundReadContentEnd_o => outboundReadContentEnd,
             outboundReadContentData_o => inFrameOutbound
         );

IoLinkBufferInbound: FIFO_BUFFER
  PORT MAP(
    rst => inFlush,
    wr_clk  => clk,
    rd_clk  => clk,
    din  => inbIN, -- used for test
    wr_en => wr_enIN, -- used for test
    rd_en => rd_en, -- inbound
    dout => inFrameInbound,
    full => fullIN, --ignored
    wr_ack => wr_ackIN, --ignored
    empty => empty, --inbound
    valid => valid, -- inbound
    rd_data_count => openvectorcontent,
    wr_data_count => openvectorcontent
  );
IoLinkBufferOutbound: FIFO_BUFFER
  PORT MAP(
    rst => outFlush,
    wr_clk  => clk,
    rd_clk  => clk,
    din  => outFrameOutbound,
    wr_en => wr_en, -- outbound
    rd_en => rd_enOut, -- used for test
    dout => outbOUT, -- outbound
    full => full, -- outbound
    wr_ack => wr_ack, -- outbound
    empty => emptyIN, -- ignored
    valid => validIN, -- ignored
    rd_data_count => openvectorcontent,
    wr_data_count => openvectorcontent
  );
-- Clock generation

clkGeneration: process
     begin
     clk <= '1';
     wait for 10 ns;
     clk <= '0';
     wait for 10 ns;
     end process;

-- Asynchronous reset generation
areset_n <= '1' after 15 ns;
testProcedureOutbound: process
begin
-- outbound part, writing on the buffer
```

```
wait for 80 ns;
-- NORMAL TEST
-- write on buffer 1
 outboundWriteContent <= '1';
 rd_enOut <= '0';
-- mac source
outbIN<= x"11";
wait for 20 ns;
outbIN<= x"11";
wait for 20 ns;
outbIN<= x"11";
wait for 20 ns;
outbIN<= x"11";
wait for 20 ns;
outbIN<= x"11";
wait for 20 ns;
outbIN<= x"11";
wait for 20 ns;
--mac destination
outbIN<= x"22";
wait for 20 ns;
outbIN<= x"22";
wait for 20 ns;
outbIN<= x"22";
wait for 20 ns;
outbIN<= x"22";
wait for 20 ns;
outbIN<= x"22";
wait for 20 ns;
outbIN<= x"22";
wait for 20 ns;
-- eth type
outbIN<= x"08";
wait for 20 ns;
outbIN<= x"00";
wait for 20 ns;
-- payload
outbIN<= x"31";
wait for 20 ns;
outbIN<= x"32";
```

```
        wait for 20 ns;
        outbIN<= x"33";
        wait for 20 ns;
        outbIN<= x"34";
        wait for 20 ns;
        outbIN<= x"35";
        wait for 20 ns;
        outbIN<= x"36";
        wait for 20 ns;
        outbIN<= x"37";
        wait for 20 ns;
        outbIN<= x"38";
        wait for 20 ns;
        outbIN<= x"39";
        wait for 20 ns;
        outbIN<= x"3A";
        wait for 20 ns;
        outbIN<= x"3B";
        wait for 20 ns;
        -- crc
        outbIN<= x"00";
        wait for 20 ns;
        outbIN<= x"01";
        wait for 20 ns;
        outbIN<= x"02";
        wait for 20 ns;
        outbIN<= x"03";
        wait for 20 ns;
        outboundWriteContent <= '0';
        outboundWriteFrame <= '1';
        wait for 20 ns;
        outboundWriteContent <= '0';
        outboundWriteFrame <= '0';
        wait for 20 ns;
        -- read from the FIFO_BUFFER
        rd_enOut <= '1';
        wait for 500 ns;
    end process;
    testProcedureInbound: process
    begin
```

```
-- inbound part, writing in the FIFO_BUFFER
     wr_enIN <= '0';
     wait for 80 ns;
     -- empty <= '0';
     wr_enIN <= '1';
     inbIN <= x"7E";
     wait for 20 ns;
     inbIN <= x"FF";
     wait for 20 ns;
     inbIN <= x"03";
     wait for 20 ns;
     inbIN <= x"00";
     wait for 20 ns;
     inbIN <= x"21";
     wait for 20 ns;
     -- payload
     inbIN<= x"31";
     wait for 20 ns;
     inbIN<= x"32";
     wait for 20 ns;
     inbIN<= x"33";
     wait for 20 ns;
     inbIN<= x"34";
     wait for 20 ns;
     inbIN<= x"35";
     wait for 20 ns;
     inbIN<= x"36";
     wait for 20 ns;
     inbIN<= x"37";
     wait for 20 ns;
     inbIN<= x"38";
     wait for 20 ns;
     inbIN<= x"39";
     wait for 20 ns;
     inbIN<= x"3A";
     wait for 20 ns;
     inbIN<= x"3B";
     wait for 20 ns;
     -- crc
     inbIN<= x"4E";
     wait for 20 ns;
     inbIN<= x"08";
     wait for 20 ns;
     inbIN<= x"BF";
     wait for 20 ns;
     inbIN<= x"B4";
     wait for 20 ns;
     -- end byte
     inbIN<= x"7E";
     wait for 20 ns;
     wr_enIN <= '1';
     -- empty <= '1';
     wait for 20 ns;
-- read from buffer procedure
     inboundReadContent<='1';
     wait for 580 ns;
     inboundReadContent<='0';
     inboundReadFrame<= '1';
     wait for 20 ns;
     inboundReadFrame<= '0';
end process;
end architecture;
```

14.4.3 Full chain testbench code

The process that generates the data is shown as follows. The declaration of all the components is not reported for the sake of brevity. All the necessary signals declaration can be found in Section 4.4 and the connection of the components in Figure 7.6.

```
StimuliProcess: process (clk, areset_n)
begin
if (areset_n = '0') then
     stateStimuli <= STATE_IDLE;
     icTctl_i <= '0';
     icTd_i <= x"0";
     counter <= 0;
elsif (clk' event and clk = '1' )then

case stateStimuli is
when STATE_IDLE   =>icTctl_i <= '0';
                        icTd_i <= x"0";
                        stateStimuli <= STATE_PREAMBLE_1_A;
```

```
when STATE_PREAMBLE_1_A =>
                        icTctl_i <= '1';
                        icTd_i <= x"5";
                        stateStimuli <= STATE_PREAMBLE_1_B;

when STATE_PREAMBLE_1_B =>
                        icTctl_i <= '1';
                        icTd_i <= x"5";
                        stateStimuli <= STATE_PREAMBLE_2_A;

when STATE_PREAMBLE_2_A =>
                        icTctl_i <= '1';
                        icTd_i <= x"5";
                        stateStimuli <= STATE_PREAMBLE_2_B;

when STATE_PREAMBLE_2_B =>
                        icTctl_i <= '1';
                        icTd_i <= x"5";
                        stateStimuli <= STATE_PREAMBLE_3_A;

when STATE_PREAMBLE_3_A =>
                        icTctl_i <= '1';
                        icTd_i <= x"5";
                        stateStimuli <= STATE_PREAMBLE_3_B;

when STATE_PREAMBLE_3_B =>
                        icTctl_i <= '1';
                        icTd_i <= x"5";
                        stateStimuli <= STATE_PREAMBLE_4_A;

when STATE_PREAMBLE_4_A =>
                        icTctl_i <= '1';
                        icTd_i <= x"5";
                        stateStimuli <= STATE_PREAMBLE_4_B;

when STATE_PREAMBLE_4_B =>
                        icTctl_i <= '1';
                        icTd_i <= x"5";
                        stateStimuli <= STATE_PREAMBLE_5_A;
```

```
when STATE_PREAMBLE_5_A =>
                        icTctl_i <= '1';
                        icTd_i <= x"5";
                        stateStimuli <= STATE_PREAMBLE_5_B;

when STATE_PREAMBLE_5_B =>
                        icTctl_i <= '1';
                        icTd_i <= x"5";
                        stateStimuli <= STATE_PREAMBLE_6_A;

when STATE_PREAMBLE_6_A =>
                        icTctl_i <= '1';
                        icTd_i <= x"5";
                        stateStimuli <= STATE_PREAMBLE_6_B;

when STATE_PREAMBLE_6_B =>
                        icTctl_i <= '1';
                        icTd_i <= x"5";
                        stateStimuli <= STATE_PREAMBLE_7_A;

when STATE_PREAMBLE_7_A =>
                        icTctl_i <= '1';
                        icTd_i <= x"5";
                        stateStimuli <= STATE_PREAMBLE_7_B;

when STATE_PREAMBLE_7_B =>
                        icTctl_i <= '1';
                        icTd_i <= x"5";
                        stateStimuli <= STATE_PREAMBLE_8_A;

when STATE_PREAMBLE_8_A =>
                        icTctl_i <= '1';
                        icTd_i <= x"5";
                        stateStimuli <= STATE_PREAMBLE_8_B;

when STATE_PREAMBLE_8_B =>
                        icTctl_i <= '1';
                        icTd_i <= x"D";
                        stateStimuli <= STATE_DESTINATION_1_A;
```

```
when STATE_DESTINATION_1_A =>
                        icTctl_i <= '1';
                        icTd_i <= x"F";
                        stateStimuli <= STATE_DESTINATION_1_B;

when STATE_DESTINATION_1_B =>
                        icTctl_i <= '1';
                        icTd_i <= x"F";
                        stateStimuli <= STATE_DESTINATION_2_A;

when STATE_DESTINATION_2_A =>
                        icTctl_i <= '1';
                        icTd_i <= x"F";
                        stateStimuli <= STATE_DESTINATION_2_B;

when STATE_DESTINATION_2_B =>
                        icTctl_i <= '1';
                        icTd_i <= x"F";
                        stateStimuli <= STATE_DESTINATION_3_A;

when STATE_DESTINATION_3_A =>
                        icTctl_i <= '1';
                        icTd_i <= x"F";
                        stateStimuli <= STATE_DESTINATION_3_B;

when STATE_DESTINATION_3_B =>
                        icTctl_i <= '1';
                        icTd_i <= x"F";
                        stateStimuli <= STATE_DESTINATION_4_A;

when STATE_DESTINATION_4_A =>
                        icTctl_i <= '1';
                        icTd_i <= x"F";
                        stateStimuli <= STATE_DESTINATION_4_B;

when STATE_DESTINATION_4_B =>
                        icTctl_i <= '1';
                        icTd_i <= x"F";
                        stateStimuli <= STATE_DESTINATION_5_A;
```

```
when STATE_DESTINATION_5_A =>
                        icTctl_i <= '1';
                        icTd_i <= x"F";
                        stateStimuli <= STATE_DESTINATION_5_B;

when STATE_DESTINATION_5_B =>
                        icTctl_i <= '1';
                        icTd_i <= x"F";
                        stateStimuli <= STATE_DESTINATION_6_A;

when STATE_DESTINATION_6_A =>
                        icTctl_i <= '1';
                        icTd_i <= x"F";
                        stateStimuli <= STATE_DESTINATION_6_B;

when STATE_DESTINATION_6_B =>
                        icTctl_i <= '1';
                        icTd_i <= x"F";
                        stateStimuli <= STATE_SOURCE_1_A;

when STATE_SOURCE_1_A =>
                        icTctl_i <= '1';
                        icTd_i <= x"1";
                        stateStimuli <= STATE_SOURCE_1_B;

when STATE_SOURCE_1_B =>
                        icTctl_i <= '1';
                        icTd_i <= x"2";
                        stateStimuli <= STATE_SOURCE_2_A;

when STATE_SOURCE_2_A =>
                        icTctl_i <= '1';
                        icTd_i <= x"1";
                        stateStimuli <= STATE_SOURCE_2_B;

when STATE_SOURCE_2_B =>
                        icTctl_i <= '1';
                        icTd_i <= x"3";
                        stateStimuli <= STATE_SOURCE_3_A;
```

```
when STATE_SOURCE_3_A =>
                        icTctl_i <= '1';
                        icTd_i <= x"1";
                        stateStimuli <= STATE_SOURCE_3_B;

when STATE_SOURCE_3_B =>
                        icTctl_i <= '1';
                        icTd_i <= x"4";
                        stateStimuli <= STATE_SOURCE_4_A;

when STATE_SOURCE_4_A =>
                        icTctl_i <= '1';
                        icTd_i <= x"1";
                        stateStimuli <= STATE_SOURCE_4_B;

when STATE_SOURCE_4_B =>
                        icTctl_i <= '1';
                        icTd_i <= x"5";
                        stateStimuli <= STATE_SOURCE_5_A;

when STATE_SOURCE_5_A =>
                        icTctl_i <= '1';
                        icTd_i <= x"1";
                        stateStimuli <= STATE_SOURCE_5_B;

when STATE_SOURCE_5_B =>
                        icTctl_i <= '1';
                        icTd_i <= x"6";
                        stateStimuli <= STATE_SOURCE_6_A;

when STATE_SOURCE_6_A =>
                        icTctl_i <= '1';
                        icTd_i <= x"1";
                        stateStimuli <= STATE_SOURCE_6_B;

when STATE_SOURCE_6_B =>
                        icTctl_i <= '1';
                        icTd_i <= x"7";
                        stateStimuli <= STATE_ETH_1_A;
```

```
when STATE_ETH_1_A =>
                        icTctl_i <= '1';
                        icTd_i <= x"8";
                        stateStimuli <= STATE_ETH_1_B;

when STATE_ETH_1_B =>
                        icTctl_i <= '1';
                        icTd_i <= x"0";
                        stateStimuli <= STATE_ETH_2_A;

when STATE_ETH_2_A =>
                        icTctl_i <= '1';
                        icTd_i <= x"0";
                        stateStimuli <= STATE_ETH_2_B;

when STATE_ETH_2_B =>
                        icTctl_i <= '1';
                        icTd_i <= x"0";
                        stateStimuli <= STATE_PAYLOAD_1_A;

when STATE_PAYLOAD_1_A =>
                        icTctl_i <= '1';
                        icTd_i <= x"0";
                        stateStimuli <= STATE_PAYLOAD_1_B;

when STATE_PAYLOAD_1_B =>
                        icTctl_i <= '1';
                        icTd_i <= x"1";
                        stateStimuli <= STATE_PAYLOAD_2_A;

when STATE_PAYLOAD_2_A =>
                        icTctl_i <= '1';
                        icTd_i <= x"0";
                        stateStimuli <= STATE_PAYLOAD_2_B;



when STATE_PAYLOAD_2_B =>
                        icTctl_i <= '1';
                        icTd_i <= x"8";
                        stateStimuli <= STATE_PAYLOAD_3_A;
```

```
when STATE_PAYLOAD_3_A =>
                       icTctl_i <= '1';
                       icTd_i <= x"0";
                       stateStimuli <= STATE_PAYLOAD_3_B;

when STATE_PAYLOAD_3_B =>
                       icTctl_i <= '1';
                       icTd_i <= x"6";
                       stateStimuli <= STATE_PAYLOAD_4_A;

when STATE_PAYLOAD_4_A =>
                       icTctl_i <= '1';
                       icTd_i <= x"0";
                       stateStimuli <= STATE_PAYLOAD_4_B;

when STATE_PAYLOAD_4_B =>
                       icTctl_i <= '1';
                       icTd_i <= x"4";
                       stateStimuli <= STATE_PAYLOAD_5_A;

when STATE_PAYLOAD_5_A =>
                       icTctl_i <= '1';
                       icTd_i <= x"0";
                       stateStimuli <= STATE_PAYLOAD_5_B;

when STATE_PAYLOAD_5_B =>
                       icTctl_i <= '1';
                       icTd_i <= x"1";
                       stateStimuli <= STATE_PAYLOAD_6_A;

when STATE_PAYLOAD_6_A =>
                       icTctl_i <= '1';
                       icTd_i <= x"0";
                       stateStimuli <= STATE_PAYLOAD_6_B;

when STATE_PAYLOAD_6_B =>
                       icTctl_i <= '1';
                       icTd_i <= x"8";
                       stateStimuli <= STATE_PAYLOAD_7_A;
```

```
when STATE_PAYLOAD_7_A =>
                        icTctl_i <= '1';
                        icTd_i <= x"6";
                        stateStimuli <= STATE_PAYLOAD_7_B;

when STATE_PAYLOAD_7_B =>
                        icTctl_i <= '1';
                        icTd_i <= x"9";
                        stateStimuli <= STATE_PAYLOAD_8_A;

when STATE_PAYLOAD_8_A =>
                        icTctl_i <= '1';
                        icTd_i <= x"c";
                        stateStimuli <= STATE_PAYLOAD_8_B;

when STATE_PAYLOAD_8_B =>
                        icTctl_i <= '1';
                        icTd_i <= x"b";
                        stateStimuli <= STATE_PAYLOAD_9_A;

when STATE_PAYLOAD_9_A =>
                        icTctl_i <= '1';
                        icTd_i <= x"0";
                        stateStimuli <= STATE_PAYLOAD_9_B;

when STATE_PAYLOAD_9_B =>
                        icTctl_i <= '1';
                        icTd_i <= x"c";
                        stateStimuli <= STATE_CRC_1_A;

when STATE_CRC_1_A =>
                        icTctl_i <= '1';
                        icTd_i <= x"b";
                        stateStimuli <= STATE_CRC_1_B;

when STATE_CRC_1_B =>
                        icTctl_i <= '1';
                        icTd_i <= x"5";
                        stateStimuli <= STATE_CRC_2_A;
```

```
when STATE_CRC_2_A =>
                        icTctl_i <= '1';
                        icTd_i <= x"2";
                        stateStimuli <= STATE_CRC_2_B;

when STATE_CRC_2_B =>
                        icTctl_i <= '1';
                        icTd_i <= x"d";
                        stateStimuli <= STATE_CRC_3_A;

when STATE_CRC_3_A =>
                        icTctl_i <= '1';
                        icTd_i <= x"9";
                        stateStimuli <= STATE_CRC_3_B;

when STATE_CRC_3_B =>
                        icTctl_i <= '1';
                        icTd_i <= x"a";
                        stateStimuli <= STATE_CRC_4_A;

when STATE_CRC_4_A =>
                        icTctl_i <= '1';
                        icTd_i <= x"e";
                        stateStimuli <= STATE_CRC_4_B;

when STATE_CRC_4_B =>
                        icTctl_i <= '1';
                        icTd_i <= x"f";
                        stateStimuli <= STATE_ENDING;

when STATE_ENDING =>
                        icTctl_i <= '0';
                        icTd_i <= x"0";
                        stateStimuli <= STATE_IDLE;

when others => stateStimuli <= STATE_IDLE;
end case;
end if;
end process;
```

15. Acknowledgment

As this thesis marks the end of my University experience, there are many people I would like to thank for participating in this journey.

I am very grateful to my thesis supervisors at Bombardier Transportation Sweden AB, Jorge Sanchez de Nova and Javier Garçia Castaño. You gave me the opportunity to live my first working and personal experience abroad. I especially thank Jorge since he saw in me the potential to develop this project and Javier for giving me advice and sharing my enthusiasm about this project. The same goes for all my colleagues at Bombardier with whom I have spent these last eight months: Mattias Magnusson, Lars Brocke, Nader Kardouni, Viktor Drozdovskyy and Olavi Kumpulainen for the infinite patience when answering all my questions. I especially enjoyed the quality time I spent with the Italian crew: Marco Barbieri, Giuseppe Palma, and Gianpaolo Giudicianni. Special thanks goes to Roberto Giovannelli for all the coffees drunk together and the patience in supporting me during the development of the project.

During these years at Politecnico di Torino I have grown a lot overcoming the challenges that I have met, and I am now ready to cope with the ones I will soon encounter in my working life. I would like to thank all my professors for providing me with not only the knowledge but also the right mindset to be a good engineer. However, these challenges have been made easier thanks to all the colleagues that I have met during this journey. Thanks, Erica Raviola and Marialuisa Michelangeli, for always have been there from the beginning to the very end of this journey regardless the physical distance. I could not be more grateful for such great journey fellows. A special thank goes also to Tommaso Marinelli for being a very good friend. I won't forget Flavio Tanese, Pietro Inglese, Andrea Casalino, Luca Sasselli, Paolo Michelotti, Fabio Castagno, and Fabio Mongardi for all the good laughs you shared with me in the hardest semester I have experienced. I also thank Federica Barbaresco, Andrea Gerini, Francesco Marrone, Milena Andrighetti, Gemma Giliberti, and Roberta Marino for always helping me and the good time together.

To strive to achieve my academic goals and complete this master thesis experience, the constant presence of my friends and family has been fundamental. I am so grateful for having my long lasting friends Arianna Macrì, Alessia Velardi and Fabia Brustia. I grew up with you and I always will, no matter the distance. A very special thank goes to my new Swedish family: Elisa Tiozzo Fasiolo, Simone Pagliano, and Emre Işeri. I have found my second home in Solna. I would like to thank in particular Renzo Enrico and Lucia De Rossi for all the support and the warm feeling of family you have given me in all these years. My gratitude goes to Alessandro for the unconditional support and love he always gave me in all these years together. Lastly, my heartfelt thank goes to all my family for being patient with me and always been there, no matter what. In particular it is dedicated to my parents Patrizia and Luca, and my sister Matilde.

16. Ringraziamenti

Con questa tesi completo la mia esperienza universitaria, e ci sono tante persone che vorrei ringraziare per aver fatto parte di questo mio viaggio.

Ringrazio molto i miei supervisori alla Bombardier Transportation Sweden AB, Jorge Sanchez de Nova e Javier Garçia Castaño. Mi avete dato l'opportunità di vivere la mia prima esperienza lavorativa e personale all'estero. In particolare, ringrazio Jorge per aver creduto nelle mie capacità per sviluppare questo progetto e Javier per tutti i consigli e l'entusiasmo condiviso nel lavorare a questo progetto. Lo stesso si può dire per tutti i miei colleghi alla Bombardier con cui ho speso questi ultimo otto mesi: Mattias Magnusson, Lars Brocke, Nader Kardouni, Viktor Drozdovskyy ed Olavi Kumpulainen per l'infinita pazienza che hanno sempre avuto nel rispondere a tutte le mie domande. Ho particolarmente apprezzato il tempo di qualità trascorso con il gruppo degli italiani: Marco Barbieri, Giuseppe Palma e Gianpaolo Giudicianni. Un ringraziamento speciale va a Roberto Giovannelli per tutti i caffè bevuti insieme e la pazienza avuta nel sostenermi durante lo sviluppo del progetto.

Durante questi anni al Politecnico di Torino sono cresciuta molto superando le sfide che ho incontrato, e ora sono pronta ad affrontare quelle che incontrerò nella mia vita lavorativa. Vorrei ringraziare tutti i miei professori per avermi trasmesso non solo la conoscenza ma la giusta mentalità per essere un buon ingegnere. Tuttavia, queste sfide sono state rese più accessibili grazie a tutti i colleghi che ho incontrato durante questo viaggio. Grazie, Erica Raviola e Marialuisa Michelangeli, per esserci sempre state dall'inizio di questo viaggio fino alla sua fine indipendentemente dalla distanza fisica. Non avrei potuto desiderare migliori compagne di viaggio. Un ringraziamento speciale va a Tommaso Marinelli per essere un ottimo amico. Non dimenticherò Flavio Tanese, Pietro Inglese, Andrea Casalino, Luca Sasselli, Paolo Michelotti, Fabio Castagno, e Fabio Mongardi per avermi tenuto su di morale durante il semestre più difficile della mia carriera universitaria. Grazie Federica Barbaresco, Andrea Gerini, Francesco Marrone, Milena Andrighetti, Gemma Giliberti, e Roberta Marino per avermi sempre aiutato e per il tempo trascorso insieme.

Nell'inseguire i miei obiettivi accademici e completare questa esperienza di tesi, la presenza costante dei miei amici e della mia famiglia è stata fondamentale. Sono davvero grata alle mie amiche di lunga data Arianna Macrì, Alessia Velardi e Fabia Brustia. Sono cresciuta e continuerò a crescere insieme a voi. Un grazie speciale alla mia nuova famiglia svedese: Elisa Tiozzo Fasiolo, Simone Pagliano, ed Emre Işeri. Ho trovato la mia seconda casa a Solna. Vorrei ringraziare in particolare Renzo Enrico e Lucia De Rossi per tutto il sostegno e il calore familiare che mi avete trasmesso in tutti questi anni. La mia gratitudine va ad Alessandro per tutto il sostegno e amore incondizionato che mi ha sempre dato in questi anni insieme. Infine, il mio più sentito ringraziamento va alla mia famiglia, I miei genitori Patrizia e Luca e mia sorella Matilde, per la pazienza e per esserci sempre stati.