**מערכת תקשורת אלחוטית של שליטה מרחוק בטכנולוגית VLSI**

**Wireless communication system of remote control in VLSI technology**

פרויקט הנדסי

דו"ח מסכם פרויקט גמר

**הוכן לשם השלמת הדרישות לקבלת**

**תואר ראשון בהנדסה B. Sc**

**מאת**

לידור מזרחי - ת.ז: 318877610

עמרי אדלן - ת.ז: 205430788

**בהנחיית**

**מ"ר רמי ברונשטיין**

**הוגש למחלקה להנדסת חשמל ואלקטרוניקה**

**המכללה האקדמית להנדסה אשדוד**

**תאריך עברי תאריך לועזי**

**מערכת תקשורת אלחוטית של שליטה מרחוק בטכנולוגית VLSI**

**Wireless communication system of remote control in VLSI technology**

פרויקט הנדסי

דוח מסכם פרויקט גמר

**הוכן לשם מילוי דרישות חלקיות לקבלת**

**תואר ראשון בהנדסה B. Sc**

**מאת**

לידור מזרחי - ת.ז: 318877610

עמרי אדלן - ת.ז: 205430788

**בהנחיית**מ"ר רמי ברונשטיין

**הוגש למחלקה להנדסת חשמל ואלקטרוניקה**

**המכללה האקדמית להנדסה אשדוד**

**חתימת הסטודנט: \_\_\_\_\_\_\_\_\_\_\_\_\_\_ תאריך: \_\_\_\_\_\_\_\_**

**חתימת המנחה: \_\_\_\_\_\_\_\_\_\_\_\_\_\_ תאריך: \_\_\_\_\_\_\_\_**

**אישור ועדת הפרויקטים: \_\_\_\_\_\_\_\_\_ תאריך: \_\_\_\_\_\_\_\_**

**עמוד אופציונלי - הפרויקט ההנדסי נעשה בשיתוף עם שם החברה/מפעל**

**בהנחיית תוארו המקצועי ושם המנחה הנוסף – אם קיים**

**יש להכניס את העמוד בספר הסופי בלבד.**

**תחום הפרויקט: תקשורת**

**סוג הפרויקט: מעשי**

**Keywords: Control Systems, Communications**

**מילות מפתח: מערכות בקרה, תקשורת**

יש לבחור לפחות 2 מילות מפתח מבין אלה רשומות מטה:

מיקרומחשבים, מעגלים אלקטרוניים (Microelectronics)

מערכות הספק (Power Systems)

מערכות בקרה (Control Systems)

RF (Radiofrequency)

עיבוד אותות (Signal Processing)

עיבוד תמונות ווידאו (Image and Video Processing)

מערכות הספק אלקטרוניות (Electronic Power Systems, Power Electronics)

חיישנים ומכשור (Sensors and Instrumentation)

תקשורת (Communications)

[אלקטרואופטיקה](http://he.wikipedia.org/wiki/אלקטרואופטיקה) ותקשורת אופטית (Optoelectronics and Optic Communications)

לייזרים ויישומיהם (LASERs and applications)

לייזרים ברפואה (LASERs and medicine)

תקשורת לוויינית (Satellite Communications)

מערכות מגנטיות, שדות (Magnetic Systems)

הינע חשמלי (Electrical Drive)

מערכות בדיקה (Verification and Test Systems)

מיקרוגלים (Microwaves)

אנטנות (Antennas)

רובוטיקה (Robotics)

מערכות מתח גבוה (High voltage systems)

**עמוד אופציונלי – הקדשה (בספר הסופי בלבד)**

**ל………………………….**

**………………………………………………**

**………………………………………………**

**עמוד אופציונלי – הבעת תודות (בספר הסופי בלבד)**

**ל………………………….**

**………………………………………………**

**…………………..………………………….**

**……………………………………………..**

**תקציר**

מערכות שליטה מרחוק ממלאות תפקיד בתעשיות שונות כגון צבא, רפואה ותחבורה על ידי מתן שליטה ממקומות מרוחקים. פרויקט זה מתמקד ביישום מערכת שלט רחוק באמצעות כרטיסי FPGA וממשק בקרה באמצעות פיתון.

הדו"ח מתחיל בחקירה של היסודות התיאורטיים של מערכות שלט רחוק. לאחר מכן הוא מתאר את דרישות המערכת הכוללות מרחק שידור, זמני תגובה של המערכת וזיהוי שגיאות. ניתנים הסברים מפורטים לגבי הידע הדרוש לתכנון ובניית המערכת. כדי לפתח מערכת אשר יהיה אפשר להתחשב בזמני תגובה, שימוש בפולינומים של זיהוי שגיאות והחלטת מרחק שידור, יש צורך להשתמש בשתי בקרי FPGA והתקנים חיצוניים שיפורטו בהמשך.

יתרה מזאת תוצג סכמה למימוש המערכת תוך מתן הסבר על צורת פעולת המערכת, המבהירה את הפונקציונליות של כל רכיב, התרומה למערכת הכוללת ושיקולים אלגוריתמיים כדי לעמוד בדרישות שצוינו.

החלק הסופי דן בבדיקות נפרדות וסימולציות שנערכו על רכיבי מערכת בודדים והערכה מקיפה של תקינות המערכת כולה באמצעות ModelSim, תוך שימת דגש על זמני מעבר על פני המערכת.

מערכת פרויקט זה נועדה להציג את היישום המעשי של מערכות שליטה מרחוק המשתמשות בטכנולוגיית FPGA, תוך התמקדות בעמידה בקריטריונים ביצועיים ותרחישי העולם האמיתי.

**Abstract**

Remote control systems fulfill roles in various industries such as military, healthcare, and transportation by providing control from distant locations. This project focuses on implementing a remote-control system using FPGA cards and control interface through Python.

The report begins with an investigation of the theoretical foundations of remote-control systems. It then describes the system requirements including transmission distance, system response times, and error detection. Detailed explanations are provided regarding the knowledge required for system design and construction. In order to develop a system that meets response time requirements, utilizes error detection polynomials, and determines transmission distance, the use of two FPGA controllers and external devices will be detailed further.

Additionally, an implementation summary of the system will be presented, explaining the system's operation, clarifying the functionality of each component, its contribution to the overall system, and algorithmic considerations to meet the specified requirements.

The final section covers separate tests and simulations conducted on individual system components and a comprehensive evaluation of the system's integrity using ModelSim, with emphasis on system transition times.

This project system aims to demonstrate the practical application of remote-control systems utilizing FPGA technology, with a focus on meeting performance criteria and real-world scenarios.

**Contents**

[**1. Introduction** 1](#_Toc162711048)

[1.1 Communication 1](#_Toc162711049)

[1.2 Wire Communication 1](#_Toc162711050)

[1.2.1 Wireless Communication and Wireless Communication System 2](#_Toc162711051)

[1.2.2 VLSI Technology in Wireless Communication System 3](#_Toc162711052)

[1.3 FPGA 3](#_Toc162711053)

[1.4 Project Description and Requirements 4](#_Toc162711054)

[**2. Literature Review** 5](#_Toc162711055)

[2.1 FPGA 5](#_Toc162711056)

[2.2 Control System 7](#_Toc162711057)

[2.3 Serial and Parallel Communication 7](#_Toc162711058)

[2.4 UART Communication 8](#_Toc162711059)

[2.5 BiPhase 9](#_Toc162711060)

[2.6 CRC 10](#_Toc162711061)

[**3. System Structure** 11](#_Toc162711062)

[3.1 Laptop Computer 11](#_Toc162711063)

[3.1.1 Python 11](#_Toc162711064)

[3.1.1.1 Imports 12](#_Toc162711065)

[3.1.1.2 Serial Configuration 12](#_Toc162711066)

[3.1.1.3 GUI Initialization 12](#_Toc162711067)

[3.1.1.4 GUI Elements Creation 12](#_Toc162711068)

[3.1.1.5 Event Handling Functions 13](#_Toc162711069)

[3.1.1.6 CRC Calculation Function 13](#_Toc162711070)

[3.1.1.7 Main Loop 13](#_Toc162711071)

[3.1.1.8 Packet Array 13](#_Toc162711072)

[3.1.2 USB transformation TTL adapter 15](#_Toc162711073)

[3.2 Card A- short range transmission 16](#_Toc162711074)

[3.2.1 STX882: 17](#_Toc162711075)

[3.2.2 Block 1:"Uart\_tx\_Constant" 18](#_Toc162711076)

[3.2.2.1 Baud Clock Generation Process ("baud\_clock") 19](#_Toc162711077)

[3.2.2.2 Rising Edge Detection Process ("rising\_edg") 20](#_Toc162711078)

[3.2.2.3 UART Transmission Finite State Machine Process ("transmission") 21](#_Toc162711079)

[3.2.3 Block 2: "Uart\_rx" 23](#_Toc162711080)

[3.2.3.1 Baud Rate Clock Generation Process ("baud\_rate\_clk") 24](#_Toc162711081)

[3.2.3.2 Rising Edge Detection Process ("araising\_edge") 24](#_Toc162711082)

[3.2.3.3 Debouncer Received Data Process ("debouncer") 25](#_Toc162711083)

[3.2.3.4 UART Reception Finite State Machine Process ("main\_rx") 26](#_Toc162711084)

[3.2.4 Block 3:"Ram2\_x" 29](#_Toc162711085)

[3.2.5 Block 4: "BiPhase\_tx" 30](#_Toc162711086)

[3.2.5.1 Baud Rate Clock Generation Process ("main\_clk") 31](#_Toc162711087)

[3.2.5.2 Rising and Falling Edge Detection Process ("cut\_main\_clk") 32](#_Toc162711088)

[3.2.5.3 Rising Edge Detection Process ("cut\_rd") 32](#_Toc162711089)

[3.2.5.4 BiPhase Transmitter Main Finite State Machine Process ("main\_state\_machine") 32](#_Toc162711090)

[3.2.5.5 BiPhase Transmitter Side Finite State Machine Process ("side\_state\_machine") 34](#_Toc162711091)

[3.2.5.6 Shift Register Process ("shift\_data\_out") 36](#_Toc162711092)

[3.2.5.7 Data Output of BiPhase Process ("biphase\_signal\_out") 36](#_Toc162711093)

[3.2.6 Block 5: "Card\_A\_Design\_Python" 38](#_Toc162711094)

[3.3 Card B - short range transmission 39](#_Toc162711095)

[3.3.1 SRX882: 40](#_Toc162711096)

[3.3.2 Block 1: "BS\_Filter" 41](#_Toc162711097)

[3.3.2.1 Data Input From Card A Process ("signal\_in ") 41](#_Toc162711098)

[3.3.2.2 Debouncing The "sig\_filter" Vector Process ("xor\_debouncer ") 41](#_Toc162711099)

[3.3.2.3 Total Check Result Process ("main\_check ") 42](#_Toc162711100)

[3.3.2.4 Filtered Output Signal Process ("signal\_out ") 42](#_Toc162711101)

[3.3.3 Block 2: "Simple\_BS" 43](#_Toc162711102)

[3.3.3.1 Rising Falling Edge Detection Process ("cut\_bi\_phase ") 44](#_Toc162711103)

[3.3.3.2 Enable Generation Process ("ena ") 44](#_Toc162711104)

[3.3.3.3 Clock Generation Process ("clk\_00") 44](#_Toc162711105)

[3.3.3.4 Rising Edge Detection Process ("cut\_clk\_00") 45](#_Toc162711106)

[3.3.3.5 Clock Generation Process ("clk\_90") 45](#_Toc162711107)

[3.3.3.6 Rising Falling Edge Detection Process ("cut\_clk\_90") 45](#_Toc162711108)

[3.3.3.7 Samples for NRZL Creation Process ("sample\_bs") 45](#_Toc162711109)

[3.3.3.8 NRZL Creation Process ("nrzl\_creation") 45](#_Toc162711110)

[3.3.4 Block 3: "CRC8BIT" 47](#_Toc162711111)

[3.3.4.1 Rising Falling Edge Detection Process ("cut\_main\_clk ") 48](#_Toc162711112)

[3.3.4.2 Shift register header bits process ("shift\_register ") 48](#_Toc162711113)

[3.3.4.3 Crc-8 operation process ("crc8bit\_operation ") 48](#_Toc162711114)

[3.3.5 Block 4: "Data\_Organizer" 51](#_Toc162711115)

[3.3.5.1 Rising Falling Edge Detection Process ("main\_clk\_cut") 52](#_Toc162711116)

[3.3.5.2 Shift register header bits process ("shift\_register ") 52](#_Toc162711117)

[3.3.5.3 Data organizer operation process ("main\_state\_machine ") 52](#_Toc162711118)

[3.3.6 Block 5: "RGB" 55](#_Toc162711119)

[3.3.6.1 RGB operation process ("set\_leds") 56](#_Toc162711120)

[3.3.7 Block 6: "Card\_B\_Design" 59](#_Toc162711121)

[3.4 Card A- long range transmission 60](#_Toc162711122)

[3.5 Card B- long range transmission 61](#_Toc162711123)

[**4. A Set of Final Tests** 62](#_Toc162711124)

[4.1 Test Bench 1: "Uart\_tx\_Rom" 63](#_Toc162711125)

[4.2 Test Bench 2: "Uart\_rx" 65](#_Toc162711126)

[4.3 Test Bench 3: "Card\_A\_Design" 66](#_Toc162711127)

[4.4 Test Bench 4: "Test\_BS" 67](#_Toc162711128)

[4.5 Test Bench 5: "Test\_CRC\_DO" 69](#_Toc162711129)

[4.6 Test Bench 6: "Card\_B\_Design" 70](#_Toc162711130)

[**5. Conclusion** 71](#_Toc162711131)

[**6. References** 73](#_Toc162711132)

[**7. Appendices** 75](#_Toc162711133)

**List of Figures**

[Figure 1.1 Basic elements of telecommunication [1] 1](#_Toc162706180)

[Figure 1.2 Basic block diagram of communication system 3](#_Toc162706181)

[Figure 1.3 FPGA Board (top view) 4](#_Toc162706182)

**------------------------------------------------------------------------**

[Figure 2. 1 General architecture of FPGA [10] 5](#_Toc162706210)

[Figure 2. 2 General architecture of CLB [10] 6](#_Toc162706211)

[Figure 2. 3 Block diagram of a closed-loop control system [12] 7](#_Toc162706212)

[Figure 2. 4 Biphase waveforms- biphase-S [16] 9](#_Toc162706213)

**List of Tables**

[Table 1.1 Differences between wireline and wireless channels [1] 2](#_Toc162706489)

1. ****Introduction****

In this chapter we will discuss the understanding of the general background in which the system deals, a control communication system based on FPGAs controllers. The purpose of the system according to its requirements and a general description of a system.

* 1. ****Communication****

Communication has been an essential aspect of human interaction and technological systems for centuries. Before the advent of wireless communication, traditional methods heavily relied on physical mediums such as electrical conductors and optical fibers for the transfer of information. In telecommunication, the exchange of data occurred through these guided channels, establishing a network of connected points. Devices like television, radio, telephones, and mobile phones, connected through networks, facilitate widespread communication. The aim remains transferring information between geographically separated users, creating a exchange of information across distances ,whether through voice, real-world images, or digital data, harnessing the sensory capabilities of eyes and ears [1]. An example illustrating the transfer of information between users is shown in Figure 1.1:

תמונה שמכילה טקסט, תרשים, צילום מסך, קו

התיאור נוצר באופן אוטומטי

Figure 1.1 Basic elements of telecommunication [1]

* 1. Wire Communication

Optical fibers, known for their ability to transmit data through pulses of light, offer vast bandwidth, targeted user connections, low transmission loss, and extended relay distances. However, are sensitive to physical damage, potentially disrupting communication channels. additionally, are more challenging and costly to install and maintain, especially in remote or difficult-to-reach locations. Due to these hurdles, in certain cases, it is preferable to utilize wireless communication [2]. An example of comparison between wired and wireless channels is shown in Table 1.1:

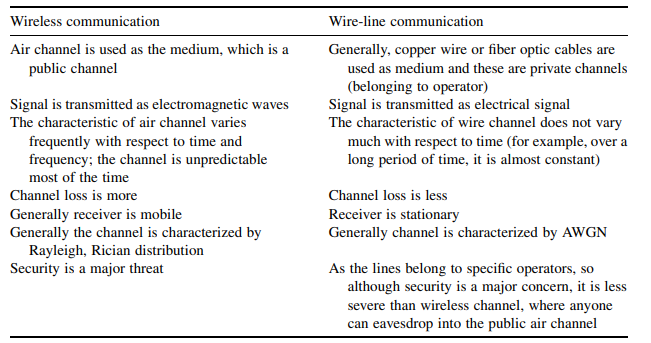


Table 1.1 Differences between wireline and wireless channels [1]

* + 1. Wireless Communication and Wireless Communication System

Wireless communication involves the transmission of information between two or more points without the need for a continuous guided medium. Instead, it harnesses electromagnetic waves, particularly radio waves, to enable communication. Radio waves are used as a means of conveying information between the transmitter and the receiver of a wireless communications system.

Simple wireless communication system consists of several key components that work together to transmit and receive information wirelessly. The transmitter component generates the information (voice, data, etc.) to be transmitted, modifies the signal to be suitable for wireless transmission. This may involve changing the frequency, amplitude, or phase of the signal and converts the electrical signal into electromagnetic waves for transmission into the air.

The channel is the air or free space through which the electromagnetic waves travel from the transmitter to the receiver. The receiver component captures the incoming electromagnetic waves, extracts the original signal from the received modulated waveform, enhances the received signal, filtering out noise and interference, decodes the information for further processing and in the end delivers the reconstructed information to the destination or end-user [1], [3]. An example illustrating the simple wireless communication system is shown in Figure 1.2:

תמונה שמכילה טקסט, צילום מסך, גופן, קו

התיאור נוצר באופן אוטומטי

Figure 1.2 Basic block diagram of communication system

* + 1. VLSI Technology in Wireless Communication System

Integrated Circuits, commonly referred to as Very Large-Scale Integration (VLSI) chips, wield significant control over an extensive array of devices in our external environment. These electronic systems are embedded within a small volume of processed silicon. They govern the functionalities of diverse technologies, ranging from IP routers managing Internet traffic, personal computers, and smartphones to the intricate components of car engines and everyday household appliances [4]. VLSI involves the integration of numerous electronic components onto a single chip, allowing for compact, energy efficient and high-performance implementations. In the context of wireless communication system, VLSI chips can be dedicated to handling tasks such as modulation, demodulation, encoding, decoding and signal processing [5], [6], [7].

* 1. FPGA

Field programmable gate array (FPGA) is a type of digital integrated circuit chip that provides reconfigurable parallel processing capability, this feature allows multiple machines to work together seamlessly in real time, thanks to powerful and flexible control functions, can be programmed in the field after production using hardware description languages such as Verilog or VHDL, and has three basic building blocks: logic gates, flip-flops + memories, and wires [8], [9]. An example of FPGA board is shown in Figure 1.3:

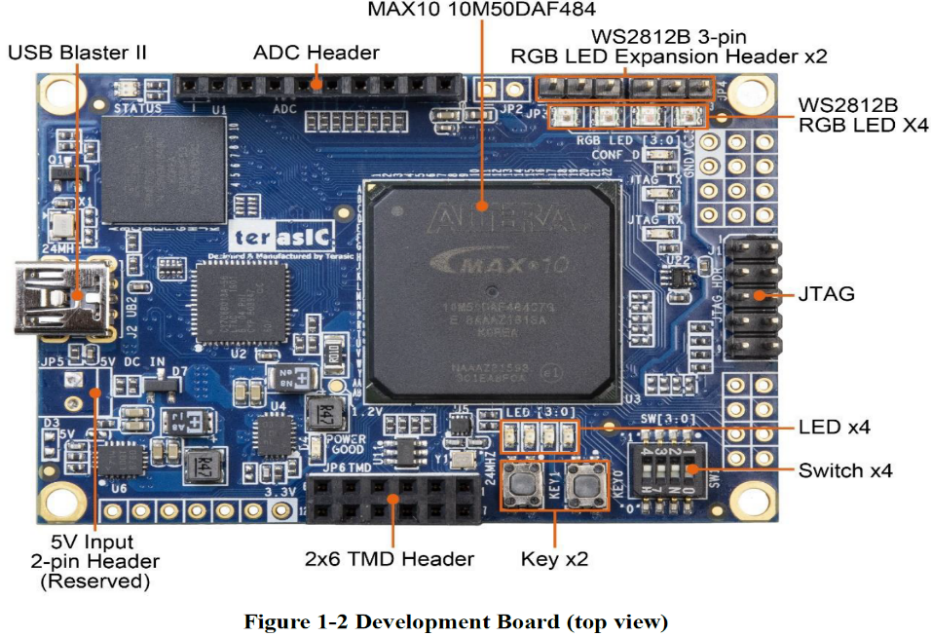


Figure 1.3 FPGA Board (top view)

* 1. Project Description and Requirements

The project is to develop a wireless remote-control system utilizing two FPGA MAX 10 cards, a laptop computer, STX882 transmitter, SRX882 receiver, CC1101 transceiver and USB transformation TTL adapter. one card will be connected to the computer in a wired manner through an adapter to receive information. The first card will wirelessly transmit the information received to the second card using the STX/SRX882 for short-distance communication and CC1101 for long-distance transmission. The received information on the second card will be showcased through LEDs and RGB LEDs, serving as a control mechanism.

The objective is to create the system described above when following the requirements below:   
A wired interface between a computer and an FPGA through communication protocol. Additionally, wireless communication is utilized for contactless data transfer between two FPGAs. Error-checking mechanisms, like CRC. A mobile interface, either computer-based or through an app, is developed for remote system control. The system will support short to long-range wireless transmission, spanning from a few meters to kilometers. Real-time operation, considering latency from both wired and wireless communication. Lastly, LEDs on the FPGAs provide visual feedback.

In the next chapter we will learn about tools for understanding the system. We will introduce FPGA controllers and their uses, we will learn about serial and parallel communication and their protocols, uses and presentation of different methods of signal decoding/encoding and error checking. In later chapters, we will implement and test the entire system.

1. Literature Review

As we mentioned, our project is built from two FPGA cards that communicate between them effectively, which simulates a remote-control system using FPGA, to understand the continuation of the project, we must get to the details of the following topics:

* 1. FPGA

FPGAs are semiconductor structures composed of a matrix of configurable logic blocks (CLBs) interconnected via programmable links. Unlike Application-Specific Integrated Circuits (ASICs), which are tailor-made for specific applications, FPGAs offer post-manufacturing reconfigurability to suit various functionalities. These devices consist of configurable logic blocks (CLBs), input/output blocks (IOBs), and interconnection networks, forming the backbone of their flexible architecture. An example of General architecture of FPGA is shown in Figure 2.1:

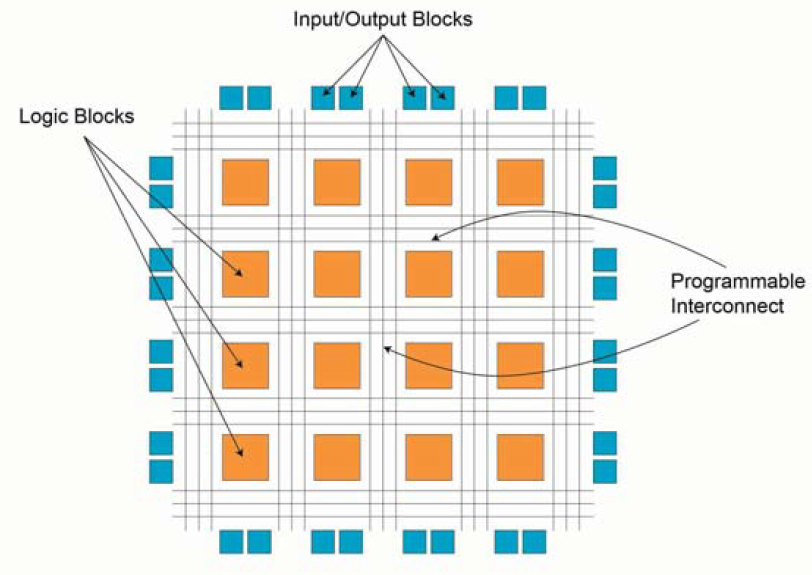


Figure 2. 1 General architecture of FPGA [10]

CLBs execute complex logic functions, implement memory functions, and synchronize code on the FPGA. They consist of flip-flops (FFs), look-up tables (LUTs), and multiplexers (MUX), enabling diverse logic implementations. FFs store logic states, LUTs provide fast output retrieval, and MUX selects inputs, collectively facilitating efficient FPGA functionality. This architectural understanding is crucial for leveraging FPGA capabilities effectively in various applications, including robotic computing. An example of General architecture of CLB is shown in Figure 2.2:

תמונה שמכילה טקסט, צילום מסך, תרשים, קו

התיאור נוצר באופן אוטומטי

Figure 2. 2 General architecture of CLB [10]

IOBs act as bridges between FPGAs and external devices, enabling seamless data and control signal exchange. Programmable Interconnects encompass a network of wires and configurable switches responsible for establishing connections between CLBs and other internal components within the FPGA.

Reprogrammability is a standout feature of FPGAs, enabling quick iteration and testing of designs without expensive fabrication processes.

Modern FPGAs come in various types - Static Random Access Memory (SRAM) is a type of semiconductor memory characterized by its fast access times and volatile nature, meaning data is retained only as long as power is applied. Organized into a matrix of rows and columns, each cell typically stores one bit of data. SRAM supports both read and write operations and field reprogrammability. Antifuse FPGAs, are one-time programmable and have complex fabrication processes. Flash-based FPGAs, while non-volatile and reprogrammable, are not recommended for runtime reconfiguration due to potential destructive effects from radiation [10], [11].

* 1. Control System

A control system, one of the fields that can be implemented with FPGA, is a mechanism designed to regulate or manage the behavior of a system. The control system monitors the output of a process and adjusts the input to maintain a desired state. This often involves a feedback loop where the system continuously compares actual output with the desired output and takes corrective actions. An example of closed loop control system block diagram is shown in figure 2.3:

תמונה שמכילה צילום מסך, טקסט, קו, גופן

התיאור נוצר באופן אוטומטי

Figure 2. 3 Block diagram of a closed-loop control system [12]

Control systems managing communication systems that utilize wave frequencies, regulate signal transmission, reception, and processing to communication. In wave frequency communication, control systems help with tasks like signal modulation, amplification, and noise reduction. They adjust parameters like frequency, phase, and amplitude for signal transmission and reception.

Transducers convert electromagnetic waves into electrical signals and vice versa, enabling the integration and manipulation of signals between the physical and electronic domains. Closed-loop control systems continuously monitor feedback signals to make real-time adjustments, ensuring the system adapts to changing conditions such as signal strength or interference [12].

* 1. Serial and Parallel Communication

Serial communication, facilitated by Universal Asynchronous Receiver-Transmitter (UART) controllers, enables bidirectional data exchange between the Systems-on-Chip and external peripherals. These controllers support full-duplex communication, allowing simultaneous transmission and reception of data. They handle the conversion of parallel data from the CPU into serial format for transmission and vice versa. Parallel communication complements serial communication by providing high-speed bidirectional data transfer capabilities. Parallel protocol controllers, such as the IEEE 1284 standard parallel protocol controller, they implement synchronous control patterns and handshake signals to enable data exchange with peripherals [13].

* 1. UART Communication

Universal Asynchronous Receiver Transmitters (UARTs), facilitate the transmission of parallel data over a serial line. Typically, UARTs are employed alongside the Recommended Standard 232 standard, is a standard defining the electrical characteristics of interfaces between devices. It's commonly used for serial communication between computers and peripherals, which defines the characteristics of data communication equipment [14]. Voltage converter chip bridge the voltage gap between RS-232 and FPGA I/O pins. A UART comprises a transmitter and receiver, with the transmitter converting parallel data into serial format and the receiver performing the reverse operation. Data transmission commences with a start bit, is the initial component of a data frame and serves as a synchronization signal for the receiving device, it always has a logical value of '0'. When data transmission begins, the start bit indicates to the receiver that a new data frame is starting, allowing it to synchronize its internal clock with the incoming data stream. Followed by data bits, are the actual information being transmitted. These bits can vary in number, and they carry the binary representation of the data being sent. An optional parity bit for error detection, and stop bits, are marks the end of a data frame and provides a brief period of time for the receiving device to prepare for the next frame, it always has a logical value of 1. The baud rate, data bits, and stop bits must be agreed upon by both the transmitter and receiver beforehand, baud rate refers to the speed at which data is transmitted over the serial communication interface. It is the number of signal changes (or symbols) per second that the UART sends or receives, the commonly used baud rates are 2400, 4800, 9600, 19200 and 38400 up to 115200 bauds. To retrieve data at the receiver end, an oversampling scheme is utilized, estimating bit middle points to synchronize data reception. This oversampling scheme forms the basis for clock signal generation, crucial for accurate data retrieval. The UART receiving subsystem comprises components for data retrieval via oversampling, which is 16, 24, and 32 for 1, 1.5, and 2 stop bits [15] .

* 1. BiPhase

Manchester encoding, a prominent biphase coding scheme. Biphase coding ensures reliable clock recovery at the receiver's end. Since it guarantees a transition in the middle of each bit period, distinguishing between '1' and '0', the receiver can accurately extract the clock signal from the data stream. This helps maintain synchronization between the transmitter and receiver, adopts alternating phases within a single square wave cycle to denote binary digits.

Biphase-S works by encoding binary data into a waveform where each bit is represented by a transition within its interval. Specifically, a transition occurs at the beginning of each bit, and for a binary 0, a second transition takes place in the middle of the bit interval, while for a binary 1, there is no second transition. This encoding method performs timing information within each bit interval, enabling synchronization between the transmitter and receiver and clock recovery. The resulting waveform contains timing components due to the transitions, and the equal polarities of 0s and 1s eliminate DC wander.

Biphase-mark code introduces transitions at the start of each bit interval, with an additional transition at the midpoint for binary 1s, aiding in timing synchronization. Differential Manchester encoding enhances reliability by utilizing transitions to represent binary values, providing immunity to polarity reversals, and improving error detection. Additionally, hybrid schemes like code mark inversion (CMI) and differential mode inversion (DMI) combine biphase coding with other techniques, offering advantages such as energy and transmission in diverse communication environments [16]. An example of Biphase waveforms - biphase-S is shown in Figure 2.3:

תמונה שמכילה טקסט, תרשים, צילום מסך, קו

התיאור נוצר באופן אוטומטי

Figure 2. 4 Biphase waveforms- biphase-S [16]

In summary, biphase coding encompasses a range of encoding techniques, each with its own waveform characteristics and advantages. Despite variations in encoding rules and waveforms, these techniques share the common goal of enabling digital communication through precise timing synchronization.

* 1. CRC

Cyclic Redundancy Check codes employ Linear Feedback Shift Registers (LFSRs) to generate a signature based on the transmitted data, which can then be used to detect any modification or corruption of bits. Serial LFSRs generate one bit of output per clock cycle, but for high-speed applications.

CRC-8 is an error-detection technique used in digital data transmission or storage. It involves generating an 8-bit checksum based on the data being processed. The process starts with an initial value, typically all zeroes, and then each bit of the data is processed sequentially. For each bit, the CRC value is XORed with the data bit, shifted right by one bit, and XORed with a predefined polynomial, if the least significant bit was 1 before the shift. After processing all the data, the resulting CRC value serves as the checksum. When the data is received or read again, the CRC calculation is repeated, and if the calculated CRC matches the received CRC, it indicates that the data was likely transmitted or stored correctly. However, a mismatch suggests that an error occurred during transmission or storage. While CRC8 is for error detection, it cannot correct errors and its effectiveness depends on the chosen polynomial.

There are several Cyclic Redundancy Check (CRC) polynomials, notable examples include:

(2.1)

(2.2)

(2.3)

(2.4)

These polynomials determine mathematical operations during CRC computation, significantly influencing error detection. When aligning the CRC polynomial with the system's specifications, if CRC8 yields an error rate sufficiently low for the system's requirements, there's no necessity to employ a higher CRC. The marginal decrease in error wouldn't justify the added complexity. Shift register operations are fundamental in hardware circuits implementing CRC computation. These operations involve shifting bits of the message and CRC register while performing XOR operations based on polynomial structures. [17], [18].

1. System Structure

In this chapter, we will delve into the structure of wireless communication remote-control system with two FPGA cards. Initially we will present the structure using a block diagram in Figure 3.1, then we will characterize each unit separately.

תמונה שמכילה טקסט, תרשים, קו, גופן

התיאור נוצר באופן אוטומטי

Fig 3.1 General view of system block diagram

System activation is initiated by initializing two "MAX10" controllers. Controller A then sends a consistent one-byte signal to the computer via a USB-TTL adapter. Our custom Python application on the PC receives this signal and then starts passing data selected at the interface we created in Python through the USB-TTL adapter back to controller A. Controller A then encodes data before transmitting it wirelessly through the STX882 transmitter. At the receiving end, the B card uses an SRX882 module to decode the transmitted data stream. After that, the decoded information is systematically arranged and displayed on an array of LEDs and an RGB display on card B.

* 1. Laptop Computer

The laptop computer is used as the control unit for the user, the software is python a very popular and useful software used in the industry.

* + 1. Python

The Python code uses libraries to establish seamless communication between the computer and the microcontroller, serving as a pivotal control interface for system operations. Moreover, the user-friendly application interface simplifies the manipulation of RGB and LED lights, ensuring swift and intuitive control.

* + - 1. Imports
* The "os" and "stat" modules are imported for handling file-related tasks, such as checking file status and attributes, though they are not utilized in this script.
* "time" is imported for time-related functions, potentially used for time delays or timing operations.
* "serial" is imported for serial communication, allowing the script to communicate with external devices, likely a microcontroller or Arduino connected via a serial port.
* "customtkinter" appears to be a custom module or library providing additional functionality or customization options for "Tkinter" GUI widgets.  
  + - 1. Serial Configuration

The script configures the serial port (COM3) with specific parameters:

* "baudrate": Sets the data transmission speed to 38400 bits per second.
* "parity": Specifies no parity checking.
* "stopbits": Sets the number of stop bits to two.
* "bytesize": Sets the number of data bits to eight.  
  + - 1. GUI Initialization

The script initializes a Tkinter window ("root") with dimensions of 800x800, providing a graphical interface for user interaction.

* + - 1. GUI Elements Creation
* Horizontal sliders ("Scale" widgets) are created to control the intensity of LEDs. Each slider corresponds to a color channel (red, green, blue) of multiple LEDs.  
  Labels ("Label" widgets) are added to provide descriptive text for the LED controls, indicating which LED or color channel each slider controls.
* Buttons ("Button" widgets) are created to allow users to toggle the state of LEDs on or off. These buttons likely correspond to individual groups of LEDs or specific functionalities.
* Custom buttons ("CTkButton" from "customtkinter") enhance the visual appearance and interactivity of the GUI, providing specific hover effects, text colors, and foreground colors.  
  + - 1. Event Handling Functions
* helloCallBack(): This function handles button clicks and toggles the state of LEDs based on the button clicked. It dynamically updates the state of LEDs based on the current state and the button clicked.
* updateArray(): This function is responsible for updating the LED control array with the values from the sliders, appending additional data (such as button states), calculating the CRC checksum for error detection, and sending the updated array via the serial port for communication with external devices.
* turnOnLed(): This function continuously loops to maintain the state of LEDs, calling updateArray() to update LED states and send data to the microcontroller via the serial port for LED control.  
  + - 1. CRC Calculation Function

"calculate\_crc8()" is function calculates an 8-bit CRC checksum for a given data array using the CRC8 polynomial (0x07) and store this checksum in the last byte in any packet, so that in CRC checking after the transmission we will get x"00" to verify a positive test result. It ensures data integrity during communication by detecting errors or corruption in the transmitted data.

* + - 1. Main Loop

The "root.mainloop()" is function starts the "Tkinter" event loop, allowing the GUI to respond to user inputs and events. It continuously monitors user interactions, updates GUI elements, and handles events such as button clicks or slider adjustments.

* + - 1. Packet Array

The Python script defines an array named "array" to encapsulate data transmitted via serial communication to a computer. It starts with header-like hexadecimal values (0x01, 0xC0, 0xCA, 0xFE, 0xAB) followed by 12 slots representing user-defined LED intensities obtained from graphical sliders. After that, there's a byte named "buttons" for button states, the first LSB bit represent 3 green LEDs when '1' logic is on. Afterward, the array is extended with 0x00 values, potentially serving as padding. Lastly, the array's final element is replaced with a CRC8 checksum computed from a subset of the array's data, ensuring data integrity during transmission. This structured array serves as a comprehensive data packet, combining user inputs, control signals, and error-checking mechanisms for reliable communication with the computer.

Overall, this Python script provides a comprehensive GUI interface for controlling LEDs connected to a microcontroller via serial communication. It combines user-friendly GUI elements with robust error detection mechanisms to ensure reliable LED control and user interaction.

* + 1. USB transformation TTL adapter

A USB to TTL adapter, also known as a USB to serial adapter, is a device that allows communication between a computer's USB port and devices that communicate using serial TTL (Transistor-Transistor Logic) signals. TTL is a type of digital signal commonly used in electronics circuits. An example of USB transformation TTL adapter in this project is shown in Figure 3.2:



Fig 3.2 USB transformation TTL adapter: SH-U09C2

In our project, the USB to TTL adapter serves as a vital intermediary, facilitating communication between the computer and the FPGA board. To ensure seamless communication, we have implemented a VHDL code for the UART protocol within the FPGA. This VHDL code enables the FPGA to interpret the serial data received from the USB to TTL adapter and to transmit data back to the computer in a format that both devices understand. The UART protocol implemented in VHDL allows for asynchronous serial communication, defining how data is framed, transmitted, and received.

We chose to use SH-U09C2 USB to TTL UART adapter is designed to facilitate communication between various devices. Its notable features include support for multiple logic levels (1.8V, 3.3V, and 5V). The MAX 10 FPGA card operates at 3.3V, which matches the voltage supported by the SH-U09C2 USB to TTL UART adapter, this voltage compatibility is crucial for ensuring proper communication between the computer and the FPGA. With both devices operating at the same voltage level, the connect is directly without needing any level-shifting circuitry. the adapter features USB 2.0 compatibility, ensuring high-speed data transfer rates of up to 3Mbps. This allows for swift and efficient communication between the adapter and connected devices. The USB-to-TTL adapter's speed, operating in the megabits per second range, far surpasses the baud rates commonly used in UART communication, such as 38400 Hz. Therefore, the adapter's speed is not a significant problem in the system's performance.

* 1. Card A- short range transmission

תמונה שמכילה טקסט, מחשב, מחשב נייד, תרשים

התיאור נוצר באופן אוטומטי

Fig 3.3 Card A short range block diagram

The information transfer route on the card A begins when the BiPhase block triggers the UART transmission control module ("Uart\_tx\_Constant"). This trigger initiates the transmission of a constant signal, "CA", which serves as a signal to the Python program running on the computer connected to Card A via UART communication. Upon receiving the "CA" signal, the Python program starts sending data packets to the UART receiver ("Uart\_rx") on Card A. These data packets are then received and processed by Card A.

The UART receiver module ("Uart\_rx") on Card A receives the data packets from the computer. It then processes the incoming data and stores it in the RAM component within Card A. This ensures that the received data is securely held within the system for further processing.

Simultaneously, the received data is encoded using the BiPhase encoding mechanism. Once encoded, the data is ready for transmission wirelessly from Card A to Card B via the BiPhase block. This wireless transmission ensures seamless communication between the two cards. The following diagram, shown in Figure 3.3, depicts the short-range Card A block diagram.

In summary, the process begins with a trigger from the BiPhase block, initiating the transmission of a constant signal ("CA") to the Python program on the computer. This signal prompts the Python program to start sending data to Card A via UART communication. The received data is then processed, stored, and encoded within Card A before being transmitted wirelessly to Card B via the "BiPhase" block.

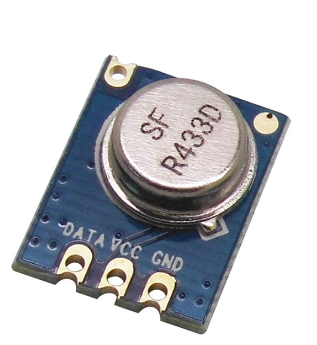
* + 1. STX882: 

Fig 3.4 STX882 transmitter

The STX882 is a wireless RF transmitter module commonly used in remote control applications. The working frequency of 433MHz, it boasts a maximum bit rate of 9600 bps, enabling seamless wireless data transmission over short distances. Typically, the module is paired with a complementary receiver module, such as the SRX882, to establish a comprehensive RF communication system capable of bidirectional data exchange. An example of STX882 transmitter in this project is shown in Figure 3.4:

The STX882 module itself does not have an encoder built into it. Instead, it relies on the input data provided to it for transmission. Typically, in projects using the STX882 module, the data to be transmitted is encoded by the microcontroller or other control circuitry before being sent to the transmitter module. This encoding could be simple, such as Manchester encoding (also known as BiPhase), or it could be more complex, such as encoding schemes used in communication protocols like UART, SPI, or custom protocol.

In our project, we opted to utilize the STX882 transmitter for short-distance transmission. This transmitter offers convenience as it does not necessitate a specific data transmission configuration. Consequently, we used BiPhase encoding, which is simpler to implement in VHDL code compared to SPI, for long-distance transmission. For readers interested in delving deeper into the technical specifications and construction of the STX882 transmitter, a link to further information is provided at the end of this book.

* + 1. Block 1:"Uart\_tx\_Constant"

In UART, data is transmitted serially bit-by-bit over a single communication line. The transmitter sends data in a predefined format, which consists of a start bit, the data bits (typically 8 bits), an optional parity bit for error checking, and a stop bit. The start bit is always a low voltage level and signals the receiver to prepare to receive data. The data bits represent the actual data being transmitted and can have a value of 0 or 1. The optional parity bit is used for error checking and can be set to odd, even, or no parity. The stop bit is always a high voltage level and signals the end of the transmission.

**Ports**

|  |  |  |  |
| --- | --- | --- | --- |
| **Description** | **Length (Bits)** | **Type** | **Name** |
| Asynchronous reset input | 1 | IN | resetn |
| System clock input (operating at 50 MHz) | 1 | IN | sysclk |
| Input signal that triggers the start of UART transmission in pulse form | 1 | IN | start\_triger |
| Output signal representing UART transmission trigger, the output bit by bit of the UART packet | 1 | OUT | uart\_tx\_triger |

**Internal signals**

|  |  |  |
| --- | --- | --- |
| **Description** | **Length (Bits)** | **Name** |
| Signal representing the current state of the finite state machine | State | state\_tx |
| Signal for generate rising edge | 1 | signal\_A\_q |
| Signal for generate rising edge | 1 | signal\_A\_q\_not |
| Signal indicating the rising edge of "sig\_baud\_clk". | 1 | sig\_arising\_edge |
| A signal to store one bit of the output "uart\_tx\_triger " | 1 | sig\_bit |
| Signal representing the baud clock 38,400Hz | 1 | sig\_baud\_clk |
| Signal representing an 8-bit data byte | 8 | sig\_byte |
| Signal representing a UART packet | 12 | sig\_packet |
| A variable used for counting | 656 | cnt\_baud |
| A variable used for counting | 16 | sig\_cntr |
| Constant representing the end bit = "111" | 3 | end\_bit |
| Constant representing the start bit = '0' | 1 | start\_bit |

This block is designed to send a set constant via UART communication to the computer. This constant, signals the authorization to begin transferring information from the computer to card A and then to card B.

**How the UART transmitter works ?**   
The UART transmitter ("Uart\_tx\_Constant") works as follows:

* + - 1. Baud Clock Generation Process ("baud\_clock")

This process generates a baud clock signal ("sig\_baud\_clk") based on the system clock ("sysclk") used for UART communication to determine the timing of UART data transmission. It counts clock cycles to produce the desired baud rate. It ensures that the baud clock is generated every 651 cycles of the system clock, every 13,020 ns (38,400Hz), and resets when "resetn" is low.

Choosing a UART frequency of 38,400Hz involves several considerations. Initially, it's crucial that the frequency is sufficiently rapid to ensure the RAM is filled with data from the computer before the transmitter can complete the transmission to the receiver. Upon calculation, it becomes evident that a frequency exceeding 4800Hz meets this requirement.

An additional crucial factor involves sending our constant, X"CA," in close proximity to the transmitter's next transmission to obtain the latest information. Therefore, it needs to be fast relative to the transmission time to ensure proximity to the subsequent transmission. Opting for a frequency of 38,400 Hz aligns with both considerations. It proves swift enough to meet the initial requirement and strikes a balance by not being excessively fast, making it a practical and well-suited choice for the application.

Explanation how to generate a clock signal using counter:

If the reset signal ("resetn") is low ('0'), it sets the "sig\_baud\_clk" to '0' and resets the counter ("cnt\_baud") to 0. If there is a rising edge on the "sysclk" signal, it increments the counter "cnt\_baud" by 1. When the counter reaches a value of 651, it toggles the "sig\_baud\_clk" signal (flips its value) and resets the counter to 0. This creates a clock signal where 651 cycles of "sysclk" is half of one cycle of this clock signal (frequency of 38,400Hz)

* + - 1. Rising Edge Detection Process ("rising\_edg")

This process detects the rising edge of the system clock and uses it to determine timing for UART transmission. It ensures synchronization with the clock signal.

Explanation how to generate a rising edge using complement:

When the reset signal ("resetn") is low ('0'), indicating a reset condition, the process sets "signal\_A\_q" to '0' and "signal\_A\_q\_not" to '1'. This is a common initialization step during a reset to ensure a known and defined state. When there is a rising edge on the "sysclk" signal (detected using the "rising\_edge" function), the process updates "signal\_A\_q" with the current value of "sig\_baud\_clk". It also updates "signal\_A\_q\_not" with the logical NOT (not) of the current value of "signal\_A\_q". This means "signal\_A\_q\_not" is the complement of "signal\_A\_q". This combination of updating "signal\_A\_q" with the current value of "sig\_baud\_clk" and updating "signal\_A\_q\_not" with the complement of "signal\_A\_q" is a common technique for capturing the rising edge of a signal. It's based on the fact that, at a rising edge, the current value of the signal is captured in one flip-flop, and the complement of that value is captured in another flip-flop.   
Finally, the logical AND of "signal\_A\_q" and "signal\_A\_q\_not" is assigned to the signal "sig\_arising\_edge". This means "sig\_arising\_edge" will be asserted ('1') only when "signal\_A\_q" is high ('1') and "signal\_A\_q\_not" is low ('0'), which occurs during a rising edge of "sig\_baud\_clk".

In summary, the rising edge is captured by updating "signal\_A\_q" with the current value of the clocked signal ("sig\_baud\_clk") and updating "signal\_A\_q\_not" with the complement of "signal\_A\_q". The logical AND of these two signals forms the output signal "sig\_arising\_edge", which indicates the occurrence of a rising edge in "sig\_baud\_clk".

* + - 1. UART Transmission Finite State Machine Process ("transmission")

The heart of the transmitter is the transmission process implements a finite state machine (FSM) for UART transmission. It progresses through various states to format and transmit data when triggered by the "start\_triger" signal. The FSM controls the generation of start and stop bits, along with data bits and transmit this packet to the computer.   
Here's a step-by-step explanation of how it works in Figure 3.5:

תמונה שמכילה תרשים, עיגול

התיאור נוצר באופן אוטומטי

**Fig 3.5 FSM transitional**

* State s0: Initially, the FSM is in state s0. When the "start\_triger" signal goes high ("start\_triger" = '1'), the FSM transitions to state s1. A transition between FSM states is on the rising edge of "sysclk"

* State s1: When the "start\_triger" signal goes low ("start\_triger" = '0'), the start trigger pulse is over and the start of UART transmission, the FSM transitions to state s2.

* State s2: In this state, a counter ("sig\_cntr") is incremented every raising edge ("sysclk"), the FSM waits for two clock cycles to generate delay, this delay period ensures that the system has sufficient time to settle into a steady state before further processing or analysis takes place, the FSM transitions to state s3.

* State s3: After two clock cycles, the FSM transitions to state s3 and prepares an 8-bit data byte ("sig\_byte") for transmission (e.g., X"CA" in this case), the FSM transitions to state s4.

* State s4: A delay is introduced to create the desired timing for UART transmission, the FSM waits for five clock cycles to generate delay, this delay period ensures that the system has sufficient time to settle into a steady state before further processing or analysis takes place, the FSM transitions to state s5.
* State s5: The UART packet ("sig\_packet") is formed by concatenating start and stop bits around the 8-bit data byte:  
  "end\_bit" & "sig\_byte" & "start\_bit" 🡺 111 & CA & 0 🡺 1 1 1 & 1 1 0 0 1 0 1 0 & 0 🡺 111110010100 = UART Packet, the FSM transitions to state s6.

* State s6: the FSM waits for one clock cycles to generate delay. the FSM progresses to state s7.

* State s7: The FSM waits for a rising edge of a clock signal ("sig\_arising\_edge") to transmit each bit of the UART packet. Data bits are transmitted from the LSB to the MSB, this operation is carried out with "Shift Register", the FSM progresses to state s8.

Explanation how to generate a left shift register:

The code checks whether a specific signal ("sig\_arising\_edge") is in an asserted state.

Shift Operation: If the condition is true, a left shift operation is performed on a signal ("sig\_packet"). The most significant bit (MSB) of "sig\_packet" is set to '1', and the rest of the bits are shifted one position to the left. Extract Bit:The least significant bit (LSB) of the original "sig\_packet" is extracted and stored in another signal ("sig\_bit"), like this 1 bit from the packet is extracted 1 by 1 from MSB to LSB. Counter Increment:A counter (e.g., "sig\_cntr") is incremented, likely to keep track of the occurrences of specific events, when reach to 12 all the packet is extracted bit by bit. In summary, the code processes a signal based on the detection of a specific condition ("sig\_arising\_edge"). When this condition is true, it involves manipulating a data signal ("sig\_packet") by shifting its bits to the left, extracting a specific bit, and updating a counter. This type of operation is commonly seen in scenarios where sequential data processing or bit manipulation is required based on specific events or conditions.

* State s8: After transmitting all bits, the FSM returns to state s0 to await the next transmission trigger.

this UART transmitter module uses a state machine to control the transmission process, generates a baud clock for timing, and constructs serial data packets for asynchronous communication. The "uart\_tx\_triger" signal serves as the output trigger for initiating UART transmission.

* + 1. Block 2: "Uart\_rx"

On the receiving end, the UART module detects the start bit and begins sampling the data at a predefined baud rate (bits per second). Once all the data bits have been received, the parity bit (if used) is checked to ensure the data is error-free. Finally, the stop bit is detected, and the UART module signals the microcontroller that the data is ready to be processed.

**Ports**

|  |  |  |  |
| --- | --- | --- | --- |
| **Description** | **Length (Bits)** | **Type** | **Name** |
| Asynchronous reset input | 1 | IN | resetn |
| System clock input (operating at 50 MHz) | 1 | IN | sysclk |
| Input signal used for addressing the RAM | 1 | IN | toggle |
| Output signal representing the UART transmission trigger | 1 | IN | detected\_bit |
| Output signal for controlling the RAM write operation. | 1 | OUT | wr\_ram |
| Output signal for addressing the RAM | 6 | OUT | ram\_address |
| Output signal for the received byte of data | 8 | OUT | detected\_byte |

**Internal signals**

|  |  |  |
| --- | --- | --- |
| **Description** | **Length (Bits)** | **Name** |
| A signal of type state, which is an enumeration representing the different states of the receiver | State | state\_rx |
| A clock signal that operates at 32 times the baud rate of 38,400 bps. It toggles every 20 "sysclk" cycles | 1 | sig\_baudx32 |
| used for debouncing the received data bit | 3 | sig\_bit |
| A debounced version of the received data bit | 1 | sig\_bouncer\_bit |
| Signal used to create a rising edge detector for "sig\_baudx32" | 1 | signal\_A\_q |
| Signal used to create a rising edge detector for "sig\_baudx32" | 1 | signal\_A\_q\_not |
| A signal that goes high on the rising edge of "sig\_baudx32", indicating the start of a bit | 1 | sig\_araising\_edge |
| A signal representing the current received data bit | 1 | sig\_data\_bit |
| A signal used to control the write operation of the RAM | 1 | sig\_wr\_ram |
| A signal used to address the RAM | 6 | sig\_ram\_address |
| A counter signal used to increment the RAM address | 5 | sig\_cnt\_address |
| A signal used to store the received byte of data | 8 | sig\_detected\_byte |

This block is designed to receive information from the computer via UART communication and transfer it smoothly to RAM storage.

**How the UART receiver works ?**   
The UART receiver ("Uart\_rx ") works as follows:

* + - 1. Baud Rate Clock Generation Process ("baud\_rate\_clk")

The "baud\_rate\_clk" process is responsible for generating a clock signal ("sig\_baudx32") with a frequency 32 times higher than the baud clock of the UART communication. This faster clock is used to sample incoming data bits with sufficient resolution. When the resetn signal is low (active-low reset), the "sig\_baudx32" signal is set to '0'. This ensures a known initial state when the system is reset, the process is sensitive to the rising edge of the system clock ("sysclk"), the "var\_cntr" variable is incremented on every rising edge of the system clock. When "var\_cntr" reaches a count of 20, the "sig\_baudx32" signal is toggled (inverted), and the counter is reset to 0 this creates a clock signal with a frequency that is 1/20th of the "sysclk" frequency, effectively creating a clock that is 32 times faster than the baud rate.

In summary, this process implements a simple counter to divide the "sysclk" frequency by 20, and on reaching the count of 20, it toggles "sig\_baudx32". This creates a clock signal with a frequency 32 times higher than the original clock. This faster clock is used for more accurate sampling of incoming UART data bits (explanation about "baud\_clock" shown in 3.2.2.1).

Baud rate clock calculation:

* + - 1. Rising Edge Detection Process ("araising\_edge")

This process detects the rising edge of the system clock ("sysclk") and uses it to determine timing for UART receiver. It ensures synchronization with the clock signal. This process creates a rising edge detector for "sig\_baudx32" using the "signal\_A\_q" and "signal\_A\_q\_not" signals (the explanation shown in 3.2.2.2).

* + - 1. Debouncer Received Data Process ("debouncer")

The debouncer process is responsible for debouncing the incoming signal "detected\_bit". Debouncing is a technique used to eliminate noise or jitter in a digital signal, ensuring a stable and reliable representation of the actual input. When the resetn signal is low (active-low reset), the "sig\_bit" vector is initialized to all zeros, and "sig\_bouncer\_bit" is set to '0', the process is sensitive to the rising edge of the system clock ("sysclk"). The "sig\_bit" vector is treated as a shift register, where the current "detected\_bit" is stored in "sig\_bit(0)", and the previous values are shifted accordingly. If all three bits in "sig\_bit" are equal, it means that the signal has remained stable for three consecutive clock cycles, in such a case, "sig\_bouncer\_bit" is assigned the value of the stable bit, effectively debouncing the signal. In summary, this debouncer process implements a simple shift register to store the last three values of the input signal "detected\_bit". If these three values are equal, the "sig\_bouncer\_bit" is set to the stable value, providing a debounced version of the input signal. This helps eliminate short-term noise or fluctuations in the signal. Example of how debouncer works is shown in Figure 3.6 and the following diagram, shown in Figure 3.7, depicts the debouncer RTL:

תמונה שמכילה טקסט, תרשים, גופן, קו

התיאור נוצר באופן אוטומטי

**Fig 3.6 Signal Before and After Debouncer**

תמונה שמכילה תרשים, תוכנית, שרטוט טכני, סכמטי

התיאור נוצר באופן אוטומטי

**Fig 3.7 Debouncer RTL**

* + - 1. UART Reception Finite State Machine Process ("main\_rx")

The heart of the receiver is the reception process implements a finite state machine (FSM) for UART reception. overseeing the reception and processing of incoming UART data, functioning as a state machine, it facilitates the sequential management of different states throughout the data reception process.   
Here's a step-by-step explanation of how it works:

* State s0: The receiver is initially in the s0 state, "sig\_wr\_ram" is set to '0', if "sig\_bouncer\_bit" (debounced signal) is '1', the bit counter ("var\_bit\_cntr") is reset, and the transmission counter ("var\_trns\_cntr") is incremented. If "var\_trns\_cntr" reaches 500,000 (10ms, A delay that ends before the transmission time), the RAM address ("sig\_ram\_address") is reset to all zeros. The chosen delay interval is carefully planned to finish before the current transmission concludes. This strategic timing enables us to reset the address before the start of the next transmission, ensuring that the RAM is ready to promptly receive the upcoming information well in advance of the next transmission. In situations where no data transfer occurs, conveyed that the incoming data defaults to a value of 1, therefore the delay is created in the scheme of 1. If "sig\_bouncer\_bit" is '0', the state transitions to s1. A transition between FSM states is on the rising edge of "sysclk"
* State s1: the receiver waits for the rising edge of the baud rate clock ("sig\_araising\_edge"), during each clock cycle, the clock counter ("var\_clk\_cntr") is incremented, When "var\_clk\_cntr" reaches 16, the sampled data bit ("sig\_data\_bit") is updated. When "var\_clk\_cntr" reaches 32, the state transitions to s2.

To ensure the accurate reception of data sent from the computer at a rate of 38,400 Hz, we will focus on the 16th bit of "sig\_bouncer\_bit", ensuring alignment with the midpoint of all transmitted data bits. The non-ideal shape of the received bits, as depicted in Fig 3.6 UART Byte Receiver, reveals a delay in the immediate drop and rise between bits. Consequently, to address this, each bit needs to be divided into several segments, and the middle bit from these segments will be extracted. This approach is chosen due to the stability of the middle section amidst uncertainties occurring at the edges between 0 and 1 bits. An example of how UART byte receiver is shown in Figure 3.8:

תמונה שמכילה טקסט, תרשים, צילום מסך, קו

התיאור נוצר באופן אוטומטי

**Fig 3.8 UART Byte Receiver**

* State s2: the receiver checks the sampled data bit ("sig\_data\_bit"), If it is '1', the state transitions back to s0. if it is '0', the state transitions to s3, and the RAM address ("sig\_ram\_address") is updated:  
    
   sig\_ram\_address <= not toggle & sig\_cnt\_address;   
    
  The "sig\_ram\_address" signal is a 6-bit vector that represents the address where the received information is stored in the RAM. The way it is manipulated, particularly with the toggle signal, indicates that the RAM has a total of 64 bytes, and the storage alternates between two halves (0-31 and 32-63) based on the state of the toggle (block BiPhase\_tx determines toggle state). This creates an alternating pattern in the most significant bit, effectively switching between the first 32 bytes and the second 32 bytes of the RAM.
* State s3: the receiver continues sampling and processing data bits, the process is similar to s1, but now it is processing the actual data bits. The sampled data bit is appended to "sig\_detected\_byte "by generate a left shift register (explanation shown in 3.2.2.3), and if "var\_bit\_cntr" reaches 8 (all bits in a byte processed), the state transitions to s4.
* State s4: the detected byte is stored in "detected\_byte" output, the address counter ("sig\_cnt\_address") is incremented to next address. The state transitions to s5.
* State s5: "sig\_wr\_ram" is set to '1', indicating that the byte in "detected\_byte" can be written to RAM. The state transitions back to s0 for the next reception.

this "main\_rx" process implements a state machine for UART reception. It detects the start bit, samples, and processes each data bit, reconstructs bytes, updates the RAM address, and signals when to write data to RAM. The process handles various states, ensuring proper reception and processing of UART data. The following diagrams, shown in Figure 3.9 and 3.10, depicts the algorithm transitional schematic diagram:

תמונה שמכילה תרשים, תוכנית, שרטוט טכני, קו

התיאור נוצר באופן אוטומטי

**Fig 3.9 "Uart\_rx" algorithm schematic part 1**

תמונה שמכילה תרשים, טקסט, תוכנית, שרטוט טכני

התיאור נוצר באופן אוטומטי

**Fig 3.10 "Uart\_rx" algorithm schematic part 2**

* + 1. **Block 3:"Ram2\_x"**

RAM (Random Access Memory) contains [multiplexing](https://en.wikipedia.org/wiki/Multiplexer) and [demultiplexing](https://en.wikipedia.org/wiki/Demultiplexing) circuitry, to connect the data lines to the addressed storage for reading or writing the entry. The RAM block is designed using Quartus software, first in the software we defined the type of data transfer into the memory, for the RAM we chose we determined that it could receive data and send data according to the voltage level at the "rden" input. After that we defined the size of the information for each address in RAM in the "data" input and finally we determined the number of RAM address cells in the "rdaddress" input.

**Ports**

|  |  |  |  |
| --- | --- | --- | --- |
| **Description** | **Length (Bits)** | **Type** | **Name** |
| The clock signal is likely used to synchronize read and write operations | 1 | IN | clock |
| The input data that can be written into the RAM during write operations | 8 | IN | data |
| This input represents the address in the memory array from which you want to read data | 6 | IN | rdaddress |
| Read enable input, when asserted it allows reading from the specified address | 1 | IN | rden |
| This input represents the address in the memory array where you want to write data | 6 | IN | wraddress |
| Write enable input, when asserted it enables writing data to the specified write address | 1 | IN | wren |
| The output data bus, which provides the data read from the specified read address | 8 | OUT | q |

The module has 64 addresses labeled from 0 to 63, with each address storing 8 bits of data. Data from the "Uart\_rx" block is sent to the RAM, which then transfers it to the "BiPhase" block. Here's how it works: initially, "Uart\_rx" sends data to the first 32 addresses, while simultaneously, data from the other 32 addresses is sent to "BiPhase." Then, the roles switch, with the addresses now sending and receiving data in reverse. This cycle repeats to transmit a total of 32 data packets, each containing 8 bits.

The purpose of the "Ram2\_X" block is crucial. It stores the data received from "Uart\_rx" and forwards it to "BiPhase" during transmission. Without "Ram2\_X," data could be lost because "Uart\_rx" operates at a much faster rate (38400 Hz), where each bit is sent in 26 microseconds, compared to "BiPhase," which takes 328 microseconds to transfer each bit. This speed difference means that "Uart\_rx" could potentially send around 12 bits before "BiPhase" manages to receive them, resulting in data loss. Therefore, "Ram2\_X" acts as a buffer, saving the incoming data until "BiPhase" is ready to receive it.

* + 1. Block 4: "BiPhase\_tx"

**Ports**

|  |  |  |  |
| --- | --- | --- | --- |
| **Description** | **Length (Bits)** | **Type** | **Name** |
| Asynchronous reset input | 1 | IN | resetn |
| System clock input (operating at 50 MHz) | 1 | IN | sysclk |
| The data input from ram | 8 | IN | q\_data\_ram |
| Bi-phase encoded output | 1 | OUT | biPhase\_tx\_out |
| Signal indicating the start of transmission | 1 | OUT | start\_strobe\_tx |
| provides the address used for reading | 6 | OUT | read\_address |
| Signal indicating a read operation | 1 | OUT | rd |
| Toggle the MSB bit of address for read/ write mode | 1 | OUT | Toggle |

**Internal signals**

|  |  |  |
| --- | --- | --- |
| **Description** | **Length (Bits)** | **Name** |
| A signal of type state, which is an enumeration representing the different states of the "BiPhase\_tx" | State | state\_bi |
| A signal of type state, which is an enumeration representing the different states of the "side\_state\_mashine " | State | state\_mini |
| This signal used to hold a count related to the main clock operation | 14 | sig\_main |
| This signal representing output data from a RAM | 8 | sig\_q\_ram\_out |
| This signal used to represent an address for reading from RAM | 6 | sig\_read\_address |
| This is signal used for counting purposes, possibly related to address generation | 5 | sig\_read\_address\_cnt |
| This signal used for shifting data | 8 | sig\_shift\_data |
| This signal indicates the rising edge of the main clock | 1 | sig\_main\_rising\_edge |
| This signal indicates the rising edge of the read clock | 1 | sig\_rd\_rising\_edge |
| This signal indicates the falling edge of the main clock | 1 | sig\_main\_falling\_edge |
| This signal is used to control reading operations | 1 | sig\_read |
| This signal represents the output of the Bi-Phase transmitter | 1 | sig\_biPhase\_tx\_out |
| This signal represents a single bit of data from the shifted data | 1 | sig\_q\_data\_bit |
| Signal used to create a rising and falling edge detector for "sig\_main\_clk" | 1 | sig\_cut |
| Signal used to create a rising and falling edge detector for "sig\_main\_clk" | 1 | sig\_cut\_not |
| Signal used to create a rising edge detector for "sig\_read" | 1 | sig\_cut\_rd\_not |
| Signal used to create a rising edge detector for "sig\_read" | 1 | sig\_cut\_rd |
| This signal used to represent the clock derived from "sig\_main" counter | 1 | sig\_main\_clk |
| This signal used to create pulse to start the "state\_mini" | 1 | sig\_inc |
| This signal is used to separate the sending of data in RAM between reading and writing | 1 | sig\_toggle |

This block is designed to fetch data from the RAM and transmit it wirelessly to the initial block of the B card, namely "BS\_Filter". It utilizes baseband communication and employs BiPhase coding for transmission.

**How the BiPhase transmitter works ?**   
The BiPhase transmitter ("BiPhase\_tx") works as follows:

* + - 1. Baud Rate Clock Generation Process ("main\_clk")

"main\_clk" process is responsible for generating a clock signal ("sig\_main\_clk") with a frequency of 3000 Hz. From the data sheet the maximum bit rate of transmitter STX-882 is 9600 bps, The transmitted baud rate is lower than the maximum rate that the STX 882 can transmit because this rate is fast enough for the requirements and allows the system to be more immune to noise and increases the transmission range. When the "resetn" signal is low (active-low reset), the "sig\_main" signal is set to '0'. the "sig\_main" variable is incremented on every rising edge of the system clock. every 2^13 rising edge the "sig\_main\_clk" change the logic state this operation generated the baud rate clock of BiPhase.

* + - 1. Rising and Falling Edge Detection Process ("cut\_main\_clk")

This process detects the rising and the falling edge of the system clock ("sysclk") and

it is used to perform the timing operations of the processes to create a BiPhase block. This process creates a rising edge and falling edge detector for "sig\_main\_clk" using the "sig\_cut" and "sig\_cut\_not" signals (the explanation for rising edge shown in 3.2.1.2).

The falling edge operation is the same as rising edge with other logic design and result :  
   
The "sig\_main\_falling\_edge" is assigned the logical complement of the logical OR operation between "sig\_cut" and "sig\_cut\_not". This signal will be high ('1') only when "sig\_cut" transitions from '1' to '0', indicating a falling edge.

|  |  |  |
| --- | --- | --- |
| sig\_cut | sig\_cut\_not | sig\_main\_falling\_edge |
| 1 | 0 | 0 |
| 0 | 1 | 1 |

In summary, this code detects both rising and falling edges of the system clock ("sysclk") and generates corresponding edge detection signals ("sig\_main\_rising\_edge" and "sig\_main\_falling\_edge"). These signals can be used in further logic to synchronize or trigger events based on clock edges.

* + - 1. Rising Edge Detection Process ("cut\_rd")

The "cut\_rd" process derives control signals related to reading data from "q\_data\_ram" and generates the "sig\_rd\_rising\_edge" signal (the explanation for rising edge shown in 3.2.1.2).

This code identifies the rising edge of the "sig\_read" signal concerning the system clock ("sysclk") and produces an associated edge detection signal named "sig\_rd\_rising\_edge". Given that the RAM operates within the MAX10 controller's clock domain ("sysclk"), we adapt the "sig\_read" pulse to align with "sysclk", ensuring it functions seamlessly with the RAM. Specifically, we generate a 20ns pulse synchronized with "sysclk" to drive the RAM effectively.

* + - 1. BiPhase Transmitter Main Finite State Machine Process ("main\_state\_machine")

In this process, data transfer unfolds through serial communication, organized by a finite state machine (FSM). this process role is to create the signals for synchronizing the reading of data from RAM and increasing the memory addresses. As a result, the signals produced by this process become indispensable tools for subsequent operations, ensuring the smooth flow of data throughout the system.  
Here's a step-by-step explanation of how it works:

* State s0: In state s0, the "main\_state\_machine" process is initially positioned. With each rising edge of "sig\_main\_clk", the state machine progresses forward. Within this state, the signal "sig\_inc" is established as '0', defining its initial condition. When "sig\_inc" transitions to '1', it promptly reverts to '0' within s0. This behavior ensures repeatability, aligning with the designated timing for transmitting a single byte. If "sig\_inc" is '0', the state transitions to s1.
* State s1: When the "sig\_read" signal goes from its initial state of '0' to '1', it starts the process of reading the data from the RAM unit. This allows retrieval of the data stored at the appropriate memory address. If "sig\_read" is '1', the state transitions to s2.
* State s2: In this state, "sig\_read" returns to its initial state of '0'. Upon reverting to '0', the subsequent rise will occur only after seven state transitions. This interval precisely aligns with the required timing to execute all operations related to a complete byte, ensuring readiness for the next read operation.If "sig\_read" is '0', the state transitions to s3.
* State s3 - State s6 : Modes S3 to S6 do not entail any internal operations. Instead, their role is to sustain serial communication, providing ample time until the statuses of "sig\_inc" and "sig\_read" transition to '1'.
* State s7: In the final state of this process, just before returning to s0, "sig\_inc" transitions to '1', this allows the initiation of the next state machine. The subsequent state machine role to modifying RAM addresses and determining the transmission interval for constant sent to the UART TX.

Overall, this FSM controls the process of reading data from RAM in a synchronized manner, where "sig\_read" enables data reading and "sig\_inc" controls the increment of memory addresses. The FSM cycles through states to coordinate these operations, ensuring smooth and sequential data retrieval.

* + - 1. BiPhase Transmitter Side Finite State Machine Process ("side\_state\_machine")

In this process, data transfer unfolds through serial communication, organized by a finite state machine (FSM), the process primary roles are:

* Firstly, it acts as the initiator for UART transmission, effectively signaling the UART transmitter unit to commence data transfer. This ensures precise synchronization, aligning the transmission precisely within the system's operational timeline.
* Secondly, it governs the timing of RAM address progression, ensuring appropriate intervals. This management enables seamless data retrieval from RAM.
* Furthermore, the process generates a toggle signal crucial for the UART receiver's operation. This toggle signal dictates which address (0-31 or 32-63) the BiPhase block transmits, providing control over the transmission process.

Here's a step-by-step explanation of how it works:

* State s0a: In state s0a of the "side\_state\_machine" process, the initial stage is set. With each rising edge of "sysclk", the state machine moves forward, ensuring precise synchronization. Within this state, if the signal "sig\_inc" is asserted ('1'), meaning that sufficient time has elapsed to transmit a complete byte, the state transitions to s1a. This meticulous timing ensures that the process begins only after the necessary duration has passed to transmit a full byte. The calculation of signal timings is meticulously handled within the "side\_state\_machine" process.
* State s1a: In state s1a of the "side\_state\_machine" process, the transition occurs when "sig\_inc" transitions back to '0'. This event signifies to the RAM that it's time to advance to the next memory address, as the data from the previous address has already been successfully transmitted. This synchronized action ensures efficient progression through memory addresses, facilitating the continuous flow of data transmission.
* State s2a: In state s2a of the "side\_state\_machine" process, the signal "start\_strobe\_tx" is generated. This signal is transmitted to the "Uart\_tx" unit, activating the process of transmitting the data stored within it. The signal "start\_strobe\_tx" remains high ('1') until "sig\_read\_address\_cnt" reaches a count of "00111", equivalent to seven counts 0-7( 8 rising edge) . Once this count is reached, indicating the completion of the transmission cycle, "start\_strobe\_tx" transitions from '1' back to '0', preparing the system for the next data transmission sequence. The signal time is calculated as follows:

It is feasible to send the trigger signal up to byte 28, serving as the indicator for updating the new information in the RAM closest to the commencement of the subsequent transmission cycle. This determination is guided by the following calculation: for every 4 bytes transmitted, 10 milliseconds are required. Loading all 32 bytes into the RAM also necessitates 10 milliseconds. By accounting for this timing, the trigger signal can be precisely synchronized with the data transmission process, ensuring seamless coordination between data updates and transmission cycles.

* State s3a: In state s3a of the side\_state\_machine process, the address counter is incremented with each rise of the "sig\_inc" signal, indicating the transmission of one byte, equivalent to 2.6 milliseconds. This operation is crucial for multiple reasons: firstly, it contributes to the generation of the "start\_strobe\_tx" signal discussed in state s2a. Secondly, it ensures the continued synchronization of data transfer between RAM and the BiPhase block. The address counter is designed with a size of 32 bits, accommodating values ranging from 0 to 31, effectively representing every half of the addresses within the RAM.
* State s4a: In state s4a when the counter has completed a full rotation within half of the RAM addresses, meaning the transmission of 32 bytes, the state of the "sig\_toggle" signal is altered. This signal generation is pivotal, as it is subsequently utilized in the "Uart\_rx" unit. The purpose of "sig\_toggle" is to create a demarcation between the 32 addresses in RAM. It delineates the boundary between the current 32-byte segment utilized by the project and the next 32-byte segment reserved for future operations. The calculation for "sig\_toggle" is as follows:
* State s5a: In state s5a, the final state of the process before returning to s0a, the information is output to "sig\_read\_address", which ultimately feeds into the output port "read\_address". This output data comprises 6 bits in size, representing addresses 0 to 63 of the RAM. It combines the "sig\_toggle", which creates a buffer between each series of 32 addresses, with the counter "sig\_read\_address\_cnt", allowing the progression of addresses within the same buffer. When "sig\_read\_address" is get "000000", it signifies the completion of sending all 64 bits stored in the RAM memory.
  + - 1. Shift Register Process ("shift\_data\_out")

The "shift\_data\_out" process handles the shifting of data in "sig\_shift\_data" based on the rising edge of "sysclk" and the value of "sig\_read". It either loads the data from "q\_data\_ram" or shifts "sig\_shift\_data" to the left. When the reset signal ("resetn") is low ('0'), indicating a reset condition, the signal "sig\_shift\_data" is initialized to all zeros. On each rising edge of the system clock ("sysclk"), the process checks if there's a rising edge in the "sig\_main\_rising\_edge" signal. If so, it further checks the state of the "sig\_read" signal, If "sig\_read" is high ('1'), indicating a read operation is requested, the signal "sig\_shift\_data" is updated with the data from the RAM ("q\_data\_ram"). If "sig\_read" is low ('0'), indicating no read operation, the process shifts the data in "sig\_shift\_data" one bit to the left (from bit 6 to bit 0) and appends a '0' at the LSB position. After the data is shifted or updated, the process assigns the MSB of "sig\_shift\_data" to the output signal "sig\_q\_data\_bit". This process continuously monitors the system clock ("sysclk") and updates the "sig\_shift\_data" signal based on the rising edge of "sig\_main\_rising\_edge" and the state of the "sig\_read" signal. It effectively manages data shifting and retrieval operations, ensuring that the output signal "sig\_q\_data\_bit" reflects the correct data bit.

* + - 1. Data Output of BiPhase Process ("biphase\_signal\_out")

The "biphase\_signal\_out" process generates the Bi-Phase encoded output signal "sig\_BiPhase\_tx\_out" based on the rising and falling edges of the main clock and the value of the most significant bit of "sig\_shift\_data". When the reset signal ("resetn") is low ('0'), indicating a reset condition, the output signal "sig\_BiPhase\_tx\_out" is set to '0'. On each rising edge of the system clock ("sysclk"), the process checks if there's either a rising edge in the "sig\_main\_rising\_edge" signal or if there's a falling edge in the "sig\_main\_falling\_edge" signal while the "sig\_q\_data\_bit" is '0', If either condition is met, the output signal "sig\_BiPhase\_tx\_out" is toggled (inverted), effectively implementing the BiPhase encoding. After the BiPhase encoding, the process updates the value of "sig\_BiPhase\_tx\_out. An example of BiPhase encoder wave format is shown in Figure 3.11:

תמונה שמכילה טקסט, תרשים, תוכנית, שרטוט טכני

התיאור נוצר באופן אוטומטי  
**Fig 3.11** **BiPhase Encoder Wave Format**

This process continuously monitors the system clock ("sysclk") and performs BiPhase encoding on the output signal "sig\_BiPhase\_tx\_out" based on specific conditions related to the rising and falling edges of the main signal and the state of the data bit "sig\_q\_data\_bit". This ensures that the BiPhase encoding is correctly applied to the output signal.

* + 1. Block 5: "Card\_A\_Design\_Python"

The "Card\_A\_Design\_Python" block serves as the central hub that integrates and connects various system blocks to form a hardware component focused on communication and data storage functionalities. It plays a pivotal role in linking together different modules such as UART communication, memory storage, and Bi-Phase encoding. Through its port interface, it enables the interaction with the external environment by providing inputs such as reset signals, clock signals "sysclk", and data signals "detected\_bit". Additionally, it outputs signals like "uart\_tx\_triger" and "BiPhase\_tx\_out", which are crucial for coordinating the communication process and providing the encoded data for transmission. By encompassing these interconnected modules, the "Card\_A\_Design\_Python" block effectively creates a comprehensive hardware solution tailored for communication and data storage tasks within a larger system context. The following diagram, shown in Figure 3.12, depicts the RTL connection diagram between the blocks of card A:

תמונה שמכילה תרשים, קו, טקסט, תוכנית

התיאור נוצר באופן אוטומטי

**Fig 3.12** **Connection Diagram "Card\_A\_Design\_Python"**

* 1. Card B - short range transmission

תמונה שמכילה טקסט, צילום מסך, גופן, תרשים

התיאור נוצר באופן אוטומטי

Fig 3.13 Card B short range block diagram

The Card B design commences with the system initialization triggered by the active low reset signal ("resetn"), ensuring all internal registers and components start in a known state. Incoming data, encoded in a bi-phase format and represented by the "BiPhase\_rx\_in" signal, undergoes filtering via the "BS\_Filter" component. This state aims to refine the signal, potentially smoothing transitions or removing noise. The filtered output, termed "sig\_bi\_phase\_filtered", then progresses to the "Simple\_BS" module, where additional processing occurs. This encompasses tasks such as clock recovery, where the signal is synchronized with the system clock ("sysclk"), and conversion to a non-return-to-zero-level (NRZL) data format, represented by "sig\_nrzl\_data". Optionally, if CRC (Cyclic Redundancy Check) is enabled, the "CRC8BIT" component calculates an 8-bit checksum ("sig\_crc8bit\_check") for error detection purposes, analyzing the integrity of the NRZL data. Following this, the processed data, accompanied by the CRC checksum if applicable, enters the "Data\_Orgenizer" block, where it is organized and formatted for display or further processing. This state may involve tasks such as assigning data to control LEDs ("sig\_load\_leds"), formatting data for display on green LEDs ("sig\_green\_leds") and generating signals for RGB LED display ("sig\_rgb\_leds"). Finally, the RGB component takes charge, controlling the RGB LEDs based on the organized data and control signals. Throughout this intricate process, clock signals ("sysclk") time the timing of operations, while control signals guide the flow of data through the system. Feedback signals may provide crucial status information, facilitating system monitoring and diagnostics. The following diagram, shown in Figure 3.13, depicts the card B block diagram.

* + 1. SRX882:

The SRX882 is the counterpart to the STX882; it's a wireless RF receiver module designed to work in conjunction with transmitters like the STX882 for bidirectional wireless communication. An example of SRX882 receiver in the project is shown in Figure 3.14:



Fig 3.14 SRX882 receiver

The SRX882 module does not specify a particular encoding method. Instead, it relies on the external circuitry (like a microcontroller) to interpret the received signal and decode the data. The choice of encoding method will depend on the specific application and the communication protocol.

In our project the decoding BiPhase-encoded signal received by the SRX882 module, it's essential to synchronize the receiver's clock with the signal and detect transitions between voltage levels to extract the original data bits. Additionally, incorporating CRC (Cyclic Redundancy Check) into the data processing stage adds an extra layer of reliability by calculating a checksum based on the received data bits. This allows for the detection of errors in the received data, enhancing the overall integrity of the communication system and ensuring the accuracy of the transmitted information. While the SRX882 module may have inherent sensitivity to detect weak signals, incorporating noise reduction filters can further enhance the performance and reliability of wireless communication system, particularly in challenging or noisy environments. like extend range of wireless communication system by improving the signal-to-noise ratio. This be useful where long-distance communication is required. For readers interested in delving deeper into the technical specifications and construction of the SRX882 receiver, a link to further information is provided at the end of this book.

* + 1. Block 1: "BS\_Filter"

**Ports**

|  |  |  |  |
| --- | --- | --- | --- |
| **Description** | **Length (Bits)** | **Type** | **Name** |
| Asynchronous reset input | 1 | IN | Resetn |
| System clock input (operating at 50 MHz) | 1 | IN | Sysclk |
| The input BiPhase signal from card A | 1 | IN | bi\_phase\_out |
| The filtered output signal | 1 | OUT | signal\_out |

**Internal signals**

|  |  |  |
| --- | --- | --- |
| **Description** | **Length (Bits)** | **Name** |
| A 8-bit vector used as a shift register to store the history of input signals | 8 | sig\_filter |
| Individual signals representing the XOR outputs of consecutive elements of the "sig\_filter" vector | 1 | sig\_check\_0" to "sig\_check\_6 |
| A signal used to determine the total check result based on the XOR outputs | 1 | sig\_total\_check |

This block is designed to filter the BiPhase signal from card A from the noises added during the broadcast in the air

**How the BiPhase transmitter works ?**   
The BiPhase transmitter ("BiPhase\_tx") works as follows:

* + - 1. Data Input From Card A Process ("signal\_in ")

This process is sensitive to changes in the "resetn" and "sysclk" signals. It initializes the "sig\_filter" signal to all zeros during reset and shifts the "bi\_phase\_out" input into the "sig\_filter" vector on every rising edge of the "sysclk" signal.

* + - 1. Debouncing The "sig\_filter" Vector Process ("xor\_debouncer ")

This process is responsible for debouncing the "sig\_filter" vector. During reset, it sets all the "sig\_check" signals to '1'. On every rising edge of "sysclk", it performs XOR operations on consecutive elements of "sig\_filter" to calculate the "sig\_check" signals. If will be 2 elements that different from each other's this "sig\_check" will be '1' because of the XOR operation, that mean that we have change in the "bi\_phase\_out" signal. In another case "sig\_check" will be '0' mean have no change in "bi\_phase\_out" signal.

* + - 1. Total Check Result Process ("main\_check ")

This process determines the total check result based on the "sig\_check" signals. During reset, "sig\_total\_check" is set to '1'. On each rising edge of "sysclk", if any of the "sig\_check" signals is '1', indicating a change in the input signal, "sig\_total\_check" is set to '0' if indicating a no change in the input signal "bi\_phase\_out". Otherwise, it remains '1'.

* + - 1. Filtered Output Signal Process ("signal\_out ")

This process is responsible for generating the filtered output signal. During reset, "signal\_out" is set to '0'. On each rising edge of "sysclk", if "sig\_total\_check" is '0', indicating a no change in the input signal, "signal\_out" is updated with the current value of "bi\_phase\_out".

Overall, this architecture implements a debouncing filter using XOR operations on consecutive elements of the input signal history. The filtered output signal is generated when a change in the input signal is detected, and the total check result is '0' when a no change signal has been detected and "signal\_out" is updated with the current value of "bi\_phase\_out", else indicating a change in the input signal "bi\_phase\_out" and "signal\_out" isn't updated with the current value of "bi\_phase\_out".

* + 1. Block 2: "Simple\_BS"

**Ports**

|  |  |  |  |
| --- | --- | --- | --- |
| **Description** | **Length (Bits)** | **Type** | **Name** |
| Asynchronous reset input | 1 | IN | Resetn |
| System clock input (operating at 50 MHz) | 1 | IN | Sysclk |
| The input BiPhase signal after filtering from "BS\_Filter" | 1 | IN | bi\_phase\_filtered |
| The clock output signal | 1 | OUT | main\_clk |
| The data output signal | 1 | OUT | nrzl\_data |

**Internal signals**

|  |  |  |
| --- | --- | --- |
| **Description** | **Length (Bits)** | **Name** |
| This signal used to represent the clock derived from counter. | 1 | sig\_00\_clk |
| Signal used to create a rising edge detector for "sig\_00\_clk " | 1 | sig\_00\_cut |
| Signal used to create a rising edge detector for "sig\_00\_clk " | 1 | sig\_00\_cut\_not |
| This signal indicates the rising edge of the "sig\_00\_clk" | 1 | sig\_00\_r |
| This signal used to represent the clock derived from counter. | 1 | sig\_90\_clk |
| Signal used to create a rising and falling edge detector for "sig\_90\_clk " | 1 | sig\_90\_cut |
| Signal used to create a rising and falling edge detector for "sig\_90\_clk " | 1 | sig\_90\_cut\_not |
| This signal indicates the rising edge of the " sig\_90\_clk " | 1 | sig\_90\_r |
| This signal indicates the falling edge of " sig\_90\_clk " | 1 | sig\_90\_f |
| Signal used to create a rising and falling edge detector for " sig\_bi\_phase\_filterd " | 1 | sig\_bi\_phase\_filterd\_cut |
| Signal used to create a rising and falling edge detector for " sig\_bi\_phase\_filterd " | 1 | sig\_bi\_phase\_filterd\_cut\_not |
| This signal indicates the rising edge of the  " sig\_bi\_phase\_filterd " | 1 | sig\_bi\_phase\_filterd\_r |
| This signal indicates the falling edge of  " sig\_bi\_phase\_filterd " | 1 | sig\_bi\_phase\_filterd\_f |
| This signal used to represent the enable derived from counter. | 1 | sig\_enable |
| This signal represent the samples from  " sig\_bi\_phase\_filterd " | 1 | sig\_ff\_A |
| This signal represent the samples from  " sig\_bi\_phase\_filterd " | 1 | sig\_ff\_B |
| This signal used to represent counter numbers of "sysclk" to generate clocks or pulses | 14 | Cnt\_clk |

This block is designed to receive the filtered BiPhase signal, subsequently sampling it, and executing various operations to produce both the transmission clock signal and the associated data signal.

**How the "Simple\_BS" works ?**   
The ("Simple\_BS") works as follows:

* + - 1. Rising Falling Edge Detection Process ("cut\_bi\_phase ")

This process is responsible for sampling the incoming "bi\_phase\_filtered" signal and generating synchronized versions of it ("sig\_bi\_phase\_filtered\_cut" and "sig\_bi\_phase\_filtered\_cut\_not"). It triggers on each rising edge of the "sysclk". When the "resetn" signal is asserted, indicating a reset condition, the sampled and inverted signals are initialized. Otherwise, during normal operation, the sampled and inverted versions of the input signal are updated based on the "bi\_phase\_filtered" input.

* + - 1. Enable Generation Process ("ena ")

This process generates an enable signal ("sig\_enable") after a certain number of clock cycles (12288). It ensures that certain actions in the module occur at specific times to synchronize the clock signal of the transmission ("sig\_00\_clk") to the "bi\_phase\_filtered" encoded signal. The counter "cnt\_clk" is incremented on each rising edge of "sysclk", and when it reaches the desired count, "sig\_enable" is set to '1'. The synchronization process involves sacrificing the first bit of data. Upon encountering the first logical 1 in the system activation code, the clock signal "main\_clk" is synchronized with the data signal "nrzl\_data". To eliminate the loss of the initial transmission, we devised a strategy: every packet intended for transmission will include a synchronization byte "01" before the header. This sync byte serves as a reference point, ensuring that the first transmission remains intact even after sync. By incorporating this sync byte into the packet structure, we maintain data integrity and preserve the accuracy of the information being transmitted.

* + - 1. Clock Generation Process ("clk\_00")

This process generates a clock signal ("sig\_00\_clk") that toggles between '1' and '0' every 8192 clock cycles. It helps in dividing the system clock to create timing references for specific operations. It triggers on each rising edge of "sysclk", when the counter "cnt\_clk" reaches a specific count (8192), the clock signal "sig\_00\_clk" toggles. This clock signal is used for timing purposes, this is the main clock signal of the transmission.

* + - 1. Rising Edge Detection Process ("cut\_clk\_00")

This process samples the "sig\_00\_clk" signal. It ensures that this clock signal is synchronized to the system clock ("sysclk"). It triggers on each rising edge of "sysclk", when the counter "cnt\_clk" reaches a specific count (8192), the clock signal "sig\_00\_clk" toggles. This clock signal is used for timing purposes.

* + - 1. Clock Generation Process ("clk\_90")

This process generates a clock signal ("sig\_90\_clk") that toggles between '0' and '1' every 4096 and 12288 clock cycles. It provides additional timing references for specific operations within the module. Similar to process "clk\_00", it triggers on each rising edge of "sysclk", based on the value of the counter "cnt\_clk", the clock signal "sig\_90\_clk" toggles at specific counts (4096 and 12288). The rising edge and falling edge of this signal are exactly in the middle of each half clock signal to allow for optimal sampling.

* + - 1. Rising Falling Edge Detection Process ("cut\_clk\_90")

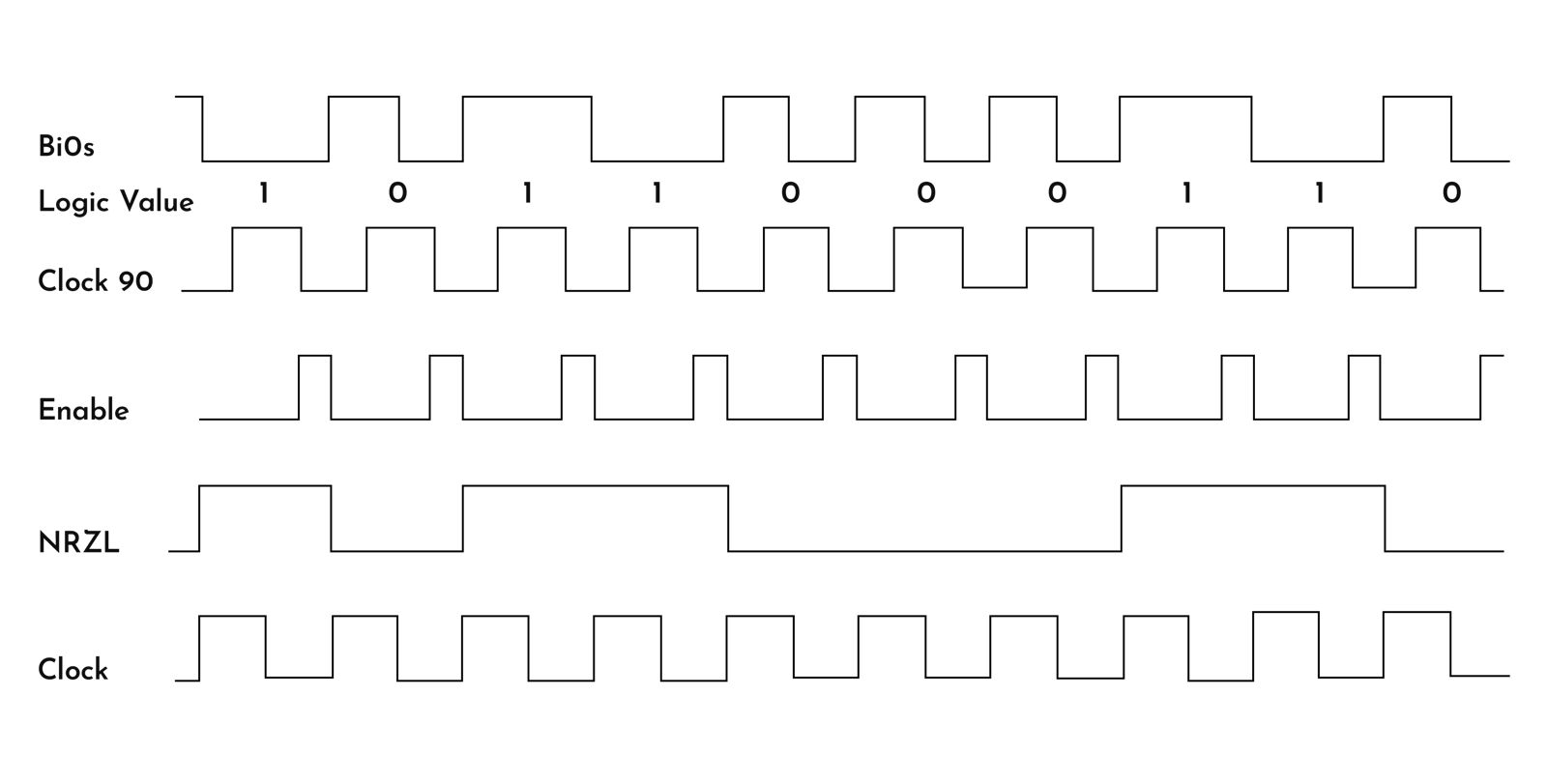
Similar to other cut processes, this one samples the "sig\_90\_clk" signal. It ensures that this clock signal is synchronized to the system clock ("sysclk"). It triggers on each rising edge of "sysclk", samples the "sig\_90\_clk" signal and updates the sampled value accordingly. Helped us to sample the "bi\_phase\_filtered" signal in the middle of every half clock for decode the data from it.

* + - 1. Samples for NRZL Creation Process ("sample\_bs")

This process samples the "bi\_phase\_filtered" signal based on the state of "sig\_00\_clk" and "sig\_90\_clk". It stores the sampled values in flip-flops ("sig\_ff\_A" and "sig\_ff\_B") based on specific conditions. Specifically, it samples "bi\_phase\_filtered" when "sig\_90\_r" is '1' and "sig\_00\_clk" is '1', and when "sig\_90\_f" is '1' and "sig\_00\_clk" is '0'. sample the middle value of "bi\_phase\_filtered" when clock pulse '1' of "sig\_00\_clk" and when clock pulse '0' of "sig\_00\_clk", then divide to 2 part the clock signal, when the first half is labeled part A, and the second half becomes part B. This helps in finding NRZ-L in the next step.

* + - 1. NRZL Creation Process ("nrzl\_creation")

This process generates the NRZ-L decoded output ("nrzl\_data") based on the sampled values of "sig\_ff\_A" and "sig\_ff\_B" when "sig\_00\_r" is asserted. It performs the final decoding step based on the sampled data. Specifically, it performs an XNOR operation between "sig\_ff\_A" and "sig\_ff\_B" when "sig\_00\_r is" '1'. An example of BiPhase Decoder Wave Format is shown in Figure 3.15:



**Fig 3.15** **BiPhase Decoder Wave Format**

Each part of the code performs specific operations necessary for processing the input signal and generating the desired output. These operations include signal sampling, clock generation, timing control, and decoding. By combining these operations in a coordinated manner, the module achieves its objective of generating NRZ-L decoded data.

* + 1. Block 3: "CRC8BIT"

**Ports**

|  |  |  |  |
| --- | --- | --- | --- |
| **Description** | **Length (Bits)** | **Type** | **Name** |
| Asynchronous reset input | 1 | IN | Resetn |
| System clock input (operating at 50 MHz) | 1 | IN | Sysclk |
| Input signal for data | 1 | IN | nrzl\_in |
| Clock signal with a period of 328 microseconds | 1 | IN | main\_clk |
| Output indicating CRC result | 1 | OUT | crc8bit\_out |

**Internal signals**

|  |  |  |
| --- | --- | --- |
| **Description** | **Length (Bits)** | **Name** |
| The state machine implementation helps organize and control the flow of the CRC calculation process | State | state\_crc |
| This signal holds the current state of the CRC value as it progresses | 8 | crc\_reg8bit |
| serves as a buffer to hold the incoming data stream | 32 | sig\_sf\_reg |
| designed to provide synchronization or timing control within your system | 1 | sig\_cut\_main\_clk |
| designed to provide synchronization or timing control within your system | 1 | sig\_cut\_main\_clk\_not |
| signal indicating the falling edge of the main clock signal "main\_clk" | 1 | sig\_main\_clk\_f |
| signal indicating the rising edge of the main clock signal "main\_clk" | 1 | sig\_main\_clk\_r |
| This counter helps in controlling the flow of the CRC calculation algorithm | 217 | cnt |

The block CRC8BIT is responsible for performing the CRC calculation. CRC (Cyclic Redundancy Check) algorithms are commonly used for error detection in digital communication systems.

**How the CRC8BIT check works ?**   
The CRC8BIT check ("CRC8BIT ") works as follows:

* + - 1. Rising Falling Edge Detection Process ("cut\_main\_clk ")

This process is responsible for sampling the incoming "main\_clk" signal and generating synchronized versions of it ("sig\_main\_clk\_r" and "sig\_in\_clk\_f"). It triggers on each rising edge of the "sysclk". "sig\_in\_clk\_f" is generated for the "shift\_register" process, handling the initial data packet (header), while "sig\_in\_clk\_r" is used for the "crc8bit\_operation" process. The generation of these signals ensures synchronization in the execution rate of operations, crucial for error detection. Further details about the generation of these signals can be found in sections 3.2.2.2 and 3.2.5.2.

* + - 1. Shift register header bits process ("shift\_register ")

The "shift\_register" process is tasked with constructing the header (X"C0CAFEAB") by shifting the "nrzl\_in" bit on every rising edge of the "sig\_main\_clk\_f" into the logic vector "sig\_sf\_reg". When the reset signal is active (resetn = '0'), the process initializes the shift register signal "sig\_sf\_reg" to all zeros, ensuring a clean state. With the rising edge of the system clock ("sysclk"), the process starts its operations. This ensures synchronization with the system clock. Within this clock domain, the process checks if the modified clock signal "sig\_main\_clk\_f" indicates a falling edge ("sig\_main\_clk\_f" = '1'). This condition ensures that the shift register updates its content at the appropriate timing. If the condition is met, the process shifts the existing contents of "sig\_sf\_reg" to the left by one bit ("sig\_sf\_reg(30 downto 0)") and adds the new input data bit "nrzl\_in" to the rightmost position of the register.

Indeed, the "shift\_register" process plays a crucial role in initializing an initial test mechanism during data transmission from CARD A to CARD B. By constructing the header (X"C0CAFEAB") through the shift of the "nrzl\_in" bit on each rising edge of "sig\_main\_clk\_f", this process initiates the test mechanism's function. If the initial test process fails, indicating an issue with the header, the CRC check will only be initiated upon receiving the specified header (X"C0CAFEAB"). Until then, the process will continue to wait for the authorized header before proceeding. This mechanism optimizes system performance by conserving time and resources that would otherwise be spent on error detection operations when unnecessary.

* + - 1. Crc-8 operation process ("crc8bit\_operation ")

In this process, data transfer unfolds through serial communication, organized by a finite state machine (FSM), the "crc8bit\_operation" process serves as the heart of the CRC-8 error detection mechanism. Here's a step-by-step explanation of how it works:

* State s0: In state s0, the process verifies the smooth reception of the header, ensuring it is error-free. If the header is successfully received without errors ("sig\_sf\_reg" =X"C0CAFEAB"), indicating a successful initial test phase, the state machine advances to stage s1. A transition between FSM states is on the rising edge of "main\_clk"
* State s1: In state s1, the CRC-8 algorithm is applied to the received data stream for error detection. The input bits are shifted into the leftmost XOR gate, representing the most significant bit (MSB) of each byte. This ensures that the MSB of each byte is processed first. Each flip-flop in the CRC register represents a single CRC output bit, with the leftmost flip-flop representing the MSB of the CRC. The flip-flops are initially cleared to zeros at the beginning of each CRC calculation, ensuring a clean state for error detection. Within the process, XOR operations are performed between the current CRC register value and the incoming data bit ("nrzl\_in"), updating each flip-flop accordingly. This process continues for each bit of the CRC register, ensuring thorough error detection across the entire data stream. Additionally, a counter ("cnt") tracks the number of iterations, incrementing with each iteration. When the counter reaches a predetermined value (216 in this case), indicating the completion of CRC calculation for the entire data stream, the state machine transitions to state s2. After transitioning, the counter is reset ("cnt" = 0) to prepare for future CRC calculations. This implementation of the CRC-8 algorithm effectively analyzes the received data stream for errors.

This polynomial represents the CRC-8 algorithm. This polynomial is used to generate the CRC checksum by performing bitwise XOR operations on the input data stream. In essence, it defines the coefficients used in the CRC computation, indicating which bits of the input data contribute to each bit of the CRC checksum. An example of CRC-8 RTL algorithm transitional diagram is shown in Figure 3.16:

תמונה שמכילה טקסט, מקלדת

התיאור נוצר באופן אוטומטי

Fig 3.16 CRC-8 RTL Algorithm Transitional Diagram

* State s2: After the completion of state s1, the CRC-8 calculation is performed, and the result is examined in stage s2. If the CRC calculation detects no errors, the value of "crc\_reg8bit" will be X"00", indicating that the CRC passed properly. In this case, the output signal "crc8bit\_out" will be set to '0'. Conversely, if errors are detected during the CRC calculation, "crc\_reg8bit" will not be X"00", and "crc8bit\_out" will be set to '1'. The CRC-8 calculation performed within the FPGA hardware is compared with an independently calculated CRC result using Python code running on a computer in the last byte in any packet. If the CRC results match, it confirms that the data is error-free, as indicated by the '0' output of "crc8bit\_out". This validation process ensures data integrity and provides confidence in the accuracy of the transmitted data.
* State s3: In state s3, following the results obtained in stage s2, the data within the CRC register ("crc\_reg8bit") is reset. This resetting process ensures that the CRC calculation starts afresh for the subsequent data that will be received, enabling accurate error detection and validation for each new data segment.
* State s4: In state s4, the system immediately transitions back to stage s0. This transition serves to introduce a small delay, allowing the system to stabilize before resuming its operation.
  + 1. Block 4: "Data\_Organizer"

**Ports**

|  |  |  |  |
| --- | --- | --- | --- |
| **Description** | **Length (Bits)** | **Type** | **Name** |
| Asynchronous reset input | 1 | IN | Resetn |
| System clock input (operating at 50 MHz) | 1 | IN | Sysclk |
| Data input signal | 1 | IN | nrzl\_data |
| Represents the main clock signal used for timing synchronization within the module | 1 | IN | main\_clk |
| Represents the result of a CRC-8 error detection calculation | 1 | IN | crc8bit\_in |
| Used to control the loading of LEDs, likely indicating when the LEDs should be updated with new data | 1 | OUT | load\_leds |
| Serves as the data signal for three green LEDs in the system | 8 | OUT | green\_leds |
| Serves as the data signal for controlling RGB LEDs in the system | 96 | OUT | rgb\_leds |

**Internal signals**

|  |  |  |
| --- | --- | --- |
| **Description** | **Length (Bits)** | **Name** |
| signal that governs the behavior and operation of the "Data\_Organizer" module | State | state\_Do |
| used to synchronize internal operations with the rising and falling edges of the "main\_clk" signal | 1 | sig\_main\_clk\_cut |
| used to synchronize internal operations with the rising and falling edges of the "main\_clk" signal | 1 | sig\_main\_clk\_cut\_not |
| represent the rising edge of the clock signal "main\_clk" | 1 | sig\_main\_clk\_r |
| represent the falling edge of the clock signal "main\_clk" | 1 | sig\_main\_clk\_f |
| serves as a buffer to hold the incoming data stream | 32 | sig\_sf\_reg |
| Used as a counter to manage the input data shifting process into the "sig\_rgb\_leds\_out" signal | 101 | sig\_cnt |
| Serves as a register to temporarily store the input data for the green LEDs | 8 | sig\_green\_leds\_reg |
| Represents the output data signal for the green LEDs | 8 | sig\_green\_leds\_out |
| Serves as the output data signal for controlling RGB LEDs | 96 | sig\_rgb\_leds\_out |

This block is designed to data organizer and CRC checker for incoming data streams. Itverifies the integrity of incoming data using a CRC8 algorithm and organizes incoming data for display on LEDs.

**How the data organizer works ?**   
The data organizer ("Data\_Organizer") works as follows:

* + - 1. Rising Falling Edge Detection Process ("main\_clk\_cut")

This process is responsible for sampling the incoming "main\_clk" signal and generating synchronized versions of it ("sig\_main\_clk\_r" and " sig\_in\_clk\_f"). It triggers on each rising edge of the "sysclk". "sig\_in\_clk\_f" is generated for the shift\_register process, handling the initial data packet (header), while "sig\_in\_clk\_r" is used for the "main\_state\_machine" process. The generation of these signals ensures synchronization in the execution rate of operations, crucial for data organizer of the green and RGB LEDS. Further details about the generation of these signals can be found in sections 3.2.1.2 and 3.2.4.2.

* + - 1. Shift register header bits process ("shift\_register ")

The "shift\_register" process is tasked with constructing the header (X"C0CAFEAB") by shifting the "nrzl\_in" bit on every rising edge of the "sig\_main\_clk\_f" into the logic vector "sig\_sf\_reg". When the reset signal is active (resetn = '0'), the process initializes the shift register signal "sig\_sf\_reg" to all zeros, ensuring a clean state. With the rising edge of the system clock ("sysclk"), the process starts its operations. This ensures synchronization with the system clock. Within this clock domain, the process checks if the modified clock signal "sig\_main\_clk\_f" indicates a falling edge ("sig\_main\_clk\_f "= '1'). This condition ensures that the shift register updates its content at the appropriate timing. If the condition is met, the process shifts the existing contents of "sig\_sf\_reg" to the left by one bit ("sig\_sf\_reg(30 downto 0)") and adds the new input data bit "nrzl\_in" to the rightmost position of the register.

The header information packet serves as a synchronization point or marker for the system. If this packet is not received or detected, it indicates that the incoming data stream may be invalid or out of sync. In such cases, attempting to process or display subsequent data, such as controlling the LEDs, would be futile and could lead to erroneous behavior.

* + - 1. Data organizer operation process ("main\_state\_machine ")

In this process, data transfer unfolds through serial communication, organized by a finite state machine (FSM), the "main\_state\_machine" process serves as the heart of the data organizer to the green and RGB LEDS. Here's a step-by-step explanation of how it works:

* State s0: In state s0, the process verifies the smooth reception of the header, ensuring it is error-free. If the header is successfully received without errors ("sig\_sf\_reg" =X"C0CAFEAB"), indicating a successful initial test phase, the state machine advances to state s1. A transition between FSM states is on the rising edge of "main\_clk"

State s1:state s1, is responsible for inserting the data into the RGB output signal using a shift register. The incoming data signal, "nrzl\_data", is shifted into the shift register with each rising of main clock. As each data bit is shifted in, a counter increment by 1. When the counter reaches 95 increments, indicating that 96 bits of data have entered "sig\_rgb\_leds\_out", the counter resets, and the process transitions to step s2. This ensures that the RGB LED data is properly assembled before proceeding to the next state. An example of RGB frame composition is shown in Figure 3.17:

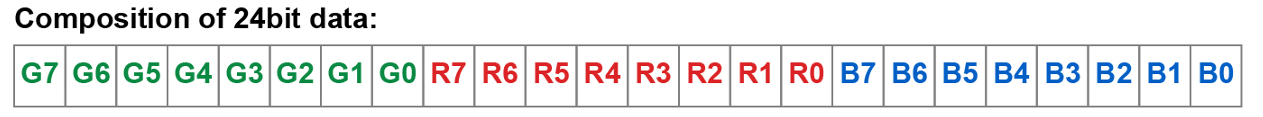


Fig 3.17 RGB Frame Composition

* State s2: In state s2, after finishing the data insertion process for the RGB LEDs in step s1. The incoming data from "nrzl\_data" is also processed to control the green LEDs. Similar to stage s1, the incoming data is shifted into a separate shift register, "sig\_green\_leds\_reg". However, in this state, the counter increments until there are 8 bits of data in "sig\_green\_leds\_reg". Once 8 bits of data have been inserted (allocation of one byte to green LEDs), the process transitions to step s3, ensuring that the data for the green LEDs is properly assembled before proceeding further.
* State s3: In state s3, delay is introduced to synchronize the system's operations before proceeding further. This pause ensures that all necessary data has been properly processed and prepared for the subsequent operations, minimizing the risk of data loss or errors due to timing mismatches. Once the delay over, the system transitions to the next state, ensuring smooth progression of operations.
* State s4: In state s4, the system precisely arranges and formats the received data to represent the status of the green LEDs within the output byte. Each bit within the "sig\_green\_leds\_out" vector corresponds directly to the status of an individual green LED, with bit positions '0' through '2' dedicated to this purpose. By organizing the data in this manner, state s4 ensures that the green LED status is accurately represented and ready for output, facilitating seamless visualization of LED status in subsequent processing stages.
* State s5: In state s5, a delay of 110 clock increments is introduced, synchronized with the main clock signal. This delay allows sufficient time for the CRC-8 block to process the incoming data and determine whether it is error-free or contains errors. The counter is incremented up to 110 to precisely control the duration of this delay, ensuring that the CRC operation completes before proceeding to the next stage of the process. The process of calculating the delay, implemented in state s5, is initiated only after the reception of a header, which occurs at the end of state s0.
* State s5a:In state s5a, the system evaluates the result of the CRC check. If the CRC result indicates that the data is error-free ("crc8bit\_in" = 0), the system proceeds to state s6. However, if the CRC result indicates errors ("crc8bit\_in" = 1), the system transitions to state s9 ,it avoids states S6 to S8, which transfer LED information to the next block. When these steps are skipped, the data isn't sent onward, means a negative result in CRC check. This decision determines the subsequent actions taken by the system based on the integrity of the received data.
* State s6-s8: In states s6 to s8, the system manages the timing for transferring the data to the RGB and green LEDs on the FPGA card. In stage s6, the signal "load\_leds" is set to 1, indicating a pause in the transmission process, allowing the system to stabilize. Then, in state s7, the most recent data is transmitted to the output ports of the RGB and green LEDs. Finally, in state s8, "load\_leds" is set to 0, signaling the start of a new transmission cycle, enabling the transfer of information to the components on the card. This sequential process ensures the orderly and synchronized transmission of data to the LEDs.
* State s9: In state s9, the system completes the state machine cycle before returning to state s0. This phase introduces a time delay to ensure the stability of the system before transitioning back to the initial state s0.
  + 1. Block 5: "RGB"

**Ports**

|  |  |  |  |
| --- | --- | --- | --- |
| **Description** | **Length (Bits)** | **Type** | **Name** |
| Connect to the asynchronous reset signal (active low) to initialize the module | 1 | IN | resetn |
| Connect to your system clock signal for synchronization | 1 | IN | sysclk |
| Control signal to load LED data | 1 | IN | load\_leds |
| Input signal for displaying data on green LEDs | 8 | IN | green\_leds |
| Output signal for displaying data on RGB LEDs | 96 | IN | rgb\_leds |
| Output signal to drive RGB LEDs' data input | 1 | OUT | OB\_LED\_RGB\_DIN |
| Output signals for specific LEDs | 1 | OUT | LED\_1 |
| Output signals for specific LEDs | 1 | OUT | LED\_2 |
| Output signals for specific LEDs | 1 | OUT | LED\_3 |

**Internal signals**

|  |  |  |
| --- | --- | --- |
| **Description** | **Length (Bits)** | **Name** |
| This signal represents the current state of the LED controller's finite state machine (FSM) | state | state\_leds |
| This signal is used for counting clock cycles or generating timing pulses | 100 | sig\_cnt |
| This signal is used to keep track of the number of iterations in a loop | 100 | sig\_cnt\_loop |
| This signal holds the RGB LED data that needs to be shifted out to the LEDs | 96 | sig\_shift\_led\_rgb |
| This signal represents the data input for the RGB LEDs | 1 | sig\_OB\_LED\_RGB\_DIN |
| This signal holds the value of a single bit from the "sig\_shift\_led\_rgb" signal | 1 | sig\_bit |

This block is designed to control RGB LEDs, facilitating the display of various colors and patterns. It manages the behavior of RGB LEDs, enabling the display of colors and patterns.

It loads data to be displayed on RGB LEDs and on specific green LEDs.

**How the RGB works ?**   
The RGB ("RGB") works as follows:

* + - 1. RGB operation process ("set\_leds")

The "set\_leds" process implements the core functionality of the RGB LED controller. It controls the state transitions, timing, and output assignments based on input signals and the system clock. By using a FSM approach within the process, the LED controller can efficiently manage the operation of RGB LEDs with precise timing and synchronization.

Here's a step-by-step explanation of how it works:

* State s0:This state primarily serves as the initialization phase or a waiting period to ensure system stability before proceeding further. By introducing a delay, it allows the system to settle into a known state, particularly after a reset event (resetn = '0'). Delaying the transition to subsequent states ensures that the system operates reliably, especially during startup or when transitioning from a reset state. A transition between FSM states is on the rising edge of "sysclk".
* State s1:"sig\_shift\_led\_rg" captured RGB LED data and "LED\_1", "LED\_2", "LED\_3" are status of individual green LEDs. This state captures the incoming RGB LED data (rgb\_leds) and extracts the status of individual green LEDs (green\_leds). RGB LED data is stored in sig\_shift\_led\_rgb, while the status of green LEDs is assigned to LED\_1, LED\_2, and LED\_3. By capturing and processing the LED data, this state prepares the necessary information for subsequent LED control operations.
* State s2:The system shifts the captured RGB LED data bits ("sig\_shift\_led\_rgb") to ensure proper alignment and synchronization for further processing. Shifting the LED data bits prepares them for subsequent operations, such as bit extraction or timing control, ensuring that the data is correctly formatted for processing.
* State s3:This state may perform additional operations on the shifted LED data or prepare for further processing steps. State s3 acts as a transitional phase, allowing the system to prepare for subsequent tasks or operations in the LED control process.
* State s4:The loop counter (sig\_cnt\_loop) is incremented to keep track of the number of bits processed. Once all 96 bits of RGB LED data have been processed, the state transitions back to s0 to restart the LED control cycle. Managing the loop counter ensures that the system processes all RGB LED data bits before restarting the LED control process, maintaining synchronization and reliability.
* State s5:"sig\_OB\_LED\_RGB\_DIN" is output signal for controlling RGB LEDs. This state controls the timing and toggling of the output signal "sig\_OB\_LED\_RGB\_DIN" based on the value of "sig\_bit." Timing control ensures that the correct signals are generated to control the behavior of the RGB LEDs, allowing for accurate representation of LED data.
* States s6 and s7:These states handle the timing control for generating signals to "sig\_OB\_LED\_RGB\_DIN" when "sig\_bit" is either '0' or '1', respectively. Timing control ensures that the correct signals are generated for "sig\_OB\_LED\_RGB\_DIN", allowing for precise control over the behavior of the RGB LEDs based on the value of "sig\_bit". An example of code timing of "sig\_OB\_LED\_RGB\_DIN" format is shown in Figure 3.18:

תמונה שמכילה תרשים, קו, גופן, תוכנית

התיאור נוצר באופן אוטומטי

Fig 3.18 Code Timing

תמונה שמכילה טקסט, גופן, צילום מסך, מספר

התיאור נוצר באופן אוטומטי

**Fig 3.19 Sequence Chart**

The logic 1 or 0 of each bit is determined as shown in Fig 3.18. It is determined by the length of the voltage level in a period. Detailed specifications can refer to Fig 3.19.

תמונה שמכילה טקסט, צילום מסך, מספר, גופן

התיאור נוצר באופן אוטומטי

**Fig 3.20 Data Transmission Method**

The data transfer protocol of the WS281 use single NRZ communication mode. As shown in Fig 3.20, after the pixel power-on reset, the DI port receive data from controller ("OB\_LED\_RGB\_DIN" output), the first pixel collectinitial 24bit data then sent to the internal data latch, the other data which reshaping by the internalsignal reshaping amplification circuit sent to the next cascade pixel through the DO port. Aftertransmission for each pixel, the signal to reduce 24bit. Pixel adopt auto reshaping transmittechnology, making the pixel cascade number is not limited the signal transmission, only dependson the speed of signal transmission.

* + 1. Block 6: "Card\_B\_Design"

The "Card\_B\_Design" block embodies the entire process of receiving, decoding, organizing, and displaying data within the system architecture. It begins by interfacing with the SRX882 module to decode the data stream wirelessly transmitted by the STX882 transmitter. Once the data is decoded, Card\_B\_Design systematically arranges it before displaying it on an array of LEDs and an RGB display. This systematic arrangement ensures that the information is presented in a clear and comprehensible manner. Additionally, Card\_B\_Design enables communication with external devices, such as computers, allowing for seamless data exchange and system control. Through its comprehensive design, Card\_B\_Design serves as the central processing unit of the system, forwards the flow of data and control signals to achieve efficient operation and accurate data presentation. The following diagram, shown in Figure 3.21, depicts the RTL connection diagram between the blocks of card B:

תמונה שמכילה קו, תרשים, מקביל, טקסט

התיאור נוצר באופן אוטומטי

**Fig 3.21** **Connection Diagram "Card\_B\_Design"**

* 1. Card A- long range transmission

תמונה שמכילה טקסט, תרשים, תוכנית, קו

התיאור נוצר באופן אוטומטי

Fig 3.19 Card A long range block diagram

* 1. Card B- long range transmission

תמונה שמכילה טקסט, תרשים, גופן, קו

התיאור נוצר באופן אוטומטי

Fig 3.20 Card B long range block diagram

1. A Set of Final Tests

In this chapter, we will present the test results of each block by using ModelSim simulation. Additionally, we will utilize a logic analyzer to provide comprehensive analysis and verification. Our approach includes conducting separate simulations for each block to assess functionality and performance, followed by summary tests to evaluate the entire project's coherence and effectiveness.

The system code is written in VHDL language , in VHDL, the library "ieee" statement signifies the utilization of the IEEE standard libraries, crucial for digital design. The "std\_logic\_1164" package, facilitating logical operations on signals represented by the "std\_logic type". Following this, the "std\_logic\_arith" package, offering arithmetic operations for "std\_logic\_vector" signals. Similarly, the "std\_logic\_unsigned" package, furnishing arithmetic operations tailored for unsigned binary numbers represented as "std\_logic\_vector". These statements collectively equip VHDL designs with fundamental functionalities for signal manipulation and arithmetic operations.

The tests were conducted on a computer equipped with an AMD Ryzen 5 5600H processor and Radeon Graphics, running at 3.30 GHz with 8GB of RAM. Each test focused on verifying the accuracy of signals and ensuring synchronization among them, as per the design in Quartus Prime software. Notably, conducting tests, especially on a personal computer, posed challenges due to lengthy processing times in ModelSim, ranging from several hours to days. To optimize efficiency, most tests were executed at higher frequencies, thereby reducing calculation times. These comprehensive tests, known as Test Benches, were conducted for every individual block or groups of blocks. For every test block, a corresponding VHDL file named "t\_X" is generated, where "X" denotes the file name. Within this file, input signals like "sysclk" or "resetn" are defined, while in other instances, additional signals such as "strobe" may also be included. These files are essential for facilitating simulations in the ModelSim software, providing a standardized setup for conducting tests and analyzing the behavior of the design under various conditions.

* 1. Test Bench 1: "Uart\_tx\_Rom"

In our simulation setup, the "Uart\_tx\_Constant" block is responsible for transmitting data sourced from a computer. However, to test the "Uart\_tx\_Constant" block independently from the computer, we introduced a separate block named "Uart\_tx\_Rom". This block is designed to retrieve data from an external ROM, simulating the data transmitted by the computer. By creating a custom code within "Uart\_tx\_Rom", we ensure that the data provided accurately emulates computer-generated data, enabling thorough testing of the "Uart\_tx\_Constant" block in isolation. An example of Uart\_tx simulation start is shown in Figure 4.1:

תמונה שמכילה צילום מסך, טקסט, תכונות מולטימדיה, תוכנה

התיאור נוצר באופן אוטומטי

**Fig 4.1** **Uart\_tx simulation start**

In the attached image, we see that the system initiates an action when the "sig\_strobe" signal goes from 1 to 0. In the "uart\_tx\_rom" testbench, we noted that this transition occurs immediately after (100ns). Following this event, data starts to propagate to the "uart\_tx\_trigger" output. Initially, the signal is set to a logical 1, and then the data transmission follows the pattern: 0 + 8 data bits + 111. It should be noted that the transmitted bit size, as observed, exactly matches the calculated value of 26000 ns, as detailed in Section 3.2.2.1 . The simulation of "Uart\_tx" data transfer is shown in Figure 4.2:

תמונה שמכילה צילום מסך, טקסט, תכונות מולטימדיה, תוכנה גרפית

התיאור נוצר באופן אוטומטי

**Fig 4.2** **"Uart\_tx" Simulation data transfer**

An example of rom .mif file with memory bytes custom are shown in Figure 4.3:

תמונה שמכילה טקסט, צילום מסך, גופן, עיצוב

התיאור נוצר באופן אוטומטי

**Fig 4.3 Rom mif file**

In Figure 4.2, the timing diagram illustrates that every 312 ns corresponds to a transition representing a start bit, 8 information bits, and 3 end bits, totaling 12 bits per transmission cycle. The data for the 8 information bits is sourced from the ROM, as depicted in Figure 4.3. Specifically, the data sequence "01" in hexadecimal corresponds to the first information, while "C0" hex corresponds to the second, and so forth.

* 1. Test Bench 2: "Uart\_rx"

To test the "uart\_rx" block, we utilize a code named "Test\_uart\_rx" that connects the three blocks: "uart\_tx\_rom", "uart\_rx" and "Ram2\_X". This code facilitates the examination of data integrity throughout the transfer process from the ROM to "uart\_tx\_Rom", then to "uart\_rx", and finally to "Ram2\_X". By executing this test setup, we can validate the data transfer pathway, ensuring the reliability of the communication system. The simulation of "Uart\_rx" data transfer is shown in Figure 4.4:

תמונה שמכילה צילום מסך, טקסט, תכונות מולטימדיה, תוכנה גרפית

התיאור נוצר באופן אוטומטי

**Fig 4.4** **"Uart\_rx" Simulation data transfer**

From the figure provided, each byte takes 307,200 nanoseconds to travel, excluding the initial stabilization period of the first "01" byte. The letter "q\_ram" represents the output of the RAM, when data is transferred sequentially, alternating between information and corresponding addresses. "detected\_byte" is used as the output of "uart\_rx", which contains all the information received from the ROM, emulating data sent from the computer. For more details on "RAM address" and "detected\_byte" functionality, see Section 3.2.3.4. This simulation demonstrates a successful implementation of serial communication operating at 38400 Hz using VHDL, which facilitates the connection between the computer and the FPGA card.

* 1. Test Bench 3: "Card\_A\_Design"

To test the "BiPhase\_tx" block and "Card\_A\_Design\_Python", we utilize a code name

"Card\_A\_Design" that connects the 4 blocks: "Uart\_tx\_Rom", "Uart\_rx", "Ram2\_X", "BiPhase\_tx". This code facilitates the examination of data integrity throughout the transfer process from the ROM to "uart\_tx", then to RAM, and then to "uart\_rx", and finally to "BiPhase\_tx". By executing this test setup, we can validate the data transfer pathway, ensuring the reliability of the communication system. The simulation of "Card\_A\_Design" data transfer to "BiPhase\_tx\_out" is shown in Figure 4.5:

תמונה שמכילה צילום מסך, טקסט, תכונות מולטימדיה, תוכנה

התיאור נוצר באופן אוטומטי

**Fig 4.5** **"Card\_A\_Design" Simulation data transfer** **to "BiPhase\_tx\_out"**

From the figure provided, each bit takes 327,680 nanoseconds to travel. The output signal "BiPhase\_tx\_out" represents the output of the "BiPhase\_tx" block. The simulation clearly demonstrates the expected behavior of the BiPhase encoding signal at the "BiPhase\_tx\_out" port. During each cycle, there's an alternating pattern of high and low states, representing logical '0's and '1's respectively. This pattern aligns with the BiPhase coding scheme, where a logical '0' is encoded as a transition between '0' and '1', while a logical '1' is represented by a steady state of either '0' or '1'. The simulation output precisely matches the expected sequence, with "01" followed by "CO" and "CA", as specified in the MIF file. This consistency confirms the accuracy of the signal encoding and decoding process, ensuring reliable data transmission within the system.

* 1. Test Bench 4: "Test\_BS"

To test the "BS\_Filter" block and "Simple\_BS", we utilize a code name "Test\_BS" that connects the 3 blocks: "Card\_A\_Design", "BS\_Filter", "Simple\_BS". This code facilitates the examination of data integrity throughout the transfer process from "Card\_A\_Design" then to "BS\_Filter" for filtering process and finally to "Simple\_BS" for decoding. Synchronization simulation between "nrzl\_data" and "main\_clk" is shown in Figure 4.6:

תמונה שמכילה צילום מסך, טקסט, תכונות מולטימדיה, תוכנה

התיאור נוצר באופן אוטומטי

**Fig 4.6 "BS\_Filter" & "Simple\_BS" Simulation sync** **"nrzl\_data" & "main\_clk"**

In the simulation, the synchronization point between the central clock signal and the information signal is discernible from the cursor's position. This synchronization occurs precisely at the juncture where the first logical 1 is sacrificed to facilitate synchronization. Subsequently, the reception of information aligns seamlessly with the desired clock signal, ensuring coordinated data transmission. Simulation of "BS\_Filter" and "Simple\_BS" data transfer as "nrzl\_data" port is shown in Figure 4.7:

תמונה שמכילה צילום מסך, טקסט, תכונות מולטימדיה, תוכנה

התיאור נוצר באופן אוטומטי

**Fig 4.7** **"BS\_Filter" & "Simple\_BS" Simulation data transfer** **to "****nrzl\_data"**

From the figure provided, each bit takes 327,680 nanoseconds to travel. The output signal "nrzl\_data" and "main\_clk" represent the outputs of the "Simple\_BS" block. The simulation results confirm the effectiveness of the "BS\_Filter" in isolating the "BiPhase\_tx\_out" signal from Card A, eliminating any noise introduced during transmission over the air. Upon decoding the coded signal, the expected information signal "nrzl\_data" is obtained, displaying the sequence "01" followed by "CO", "CA", "FE", and "AB". Furthermore, the main clock signal "main\_clk" is accurately synchronized with the information signal, validating the synchronization process. Additionally, the shifted clock signal "clk90" is generated according to the design specifications, exhibiting a 90-degree phase shift relative to the main clock signal. This ensures that each rising and falling edge of "clk90" aligns precisely with the midpoint of each half cycle of the main clock signal for sample the "BiPhase\_tx\_out" signal. The thorough verification and absence of deviations or errors underscore the meticulous planning and successful implementation of the system design.

* 1. Test Bench 5: "Test\_CRC\_DO"

To test the "CRC8BIT" block and "Data\_Organizer", we utilize a code name

"Test\_CRC\_DO" that connects the 3 blocks: "ROM\_CRC", "CRC8BIT", "Data\_Organizer". This code facilitates the examination of data integrity throughout the transfer process from "ROM\_CRC" then to "CRC8BIT" for crc checking process and finally to "Data\_Organizer" for data organize. Integrity check simulation of CRC Check and Data Organize ports are shown in Figure 4.8:

תמונה שמכילה צילום מסך, טקסט, תוכנה, תכונות מולטימדיה

התיאור נוצר באופן אוטומטי

**Fig 4.8 "CRC8BIT"&"****Data\_Organizer" Simulation** **CRC Check &** **Data Organize**

The output signals "crc\_reg8bit\_out" and "crc8bit\_out" represent the outputs of the "CRC8BIT" block and the output signals "load\_leds", "green\_leds" and "rgb\_leds" represent the outputs of the "Data\_Organizer" block. In the simulation, the CRC check, represented by the "crc\_reg8bit\_out" signal, yields a normal result ("00"), aligning with the expected outcome based on the CRC calculation manually performed and stored in the last byte of the memory according to the MIF file. Additionally, the "crc8bit\_out" signal transitions to a logical 0, signaling that the information is normal for the subsequent "Data\_Organizer" block. Upon receiving this normal CRC check result, the "load\_leds" pulse is generated, responsible the transfer of correct information to the "green\_leds" and "rgb\_leds". The simulation confirms that this transfer occurs in an orderly manner, with the information from the pre-planned memory correctly reflected in the green and RGB LEDs signals, respectively. Thus, the simulated CRC test and information organization proceed seamlessly, validating the integrity of the data processing flow and demonstrating a successful simulation outcome.

* 1. Test Bench 6: "Card\_B\_Design"

To test the " Card\_B\_Design " block, we utilize a code name "Card\_B\_Design" that connects

the 6 blocks: "BiPhase\_Generator", "BS\_Filter", "Simple\_BS", "CRC8BIT",

"Data\_Organizer", "RGB". This code facilitates the examination of data integrity throughout the transfer process from "BiPhase\_Generator" to "BS\_Filter" for filtering, through "Simple\_BS" for encoding then to "CRC8BIT" for crc checking and then "Data\_Organizer" for data organize and finally to "RGB" for manipulate the data to the LEDs. Integrity check simulation of the system output is shown in Figure 4.9:

תמונה שמכילה צילום מסך, תכונות מולטימדיה, תוכנה גרפית, תוכנה

התיאור נוצר באופן אוטומטי

**Fig 4.9 "Card\_B\_Design" Simulation,** **The System Output**

The simulation reveals that each bit transmitted to the RGB LEDs via the "OB\_LED\_RGB\_DIN" signal follows a precise timing pattern, with a duration of 1100ns per bit and the correct data is received from memory. For a bit '1', the signal exhibits a logical 1 state lasting 740ns followed by a logical 0 state lasting 360ns. Conversely, a bit '0' is characterized by a logical 0 state lasting 800ns followed by a logical 1 state lasting 300ns, consistent with the specifications outlined in the datasheet. Moreover, the state of the green LEDs corresponds accurately to the information retrieved from memory, with two LEDs illuminated and one LED turned off. This alignment between the simulated transmission and the expected behavior confirms that the information is indeed transmitted as intended, validating the integrity of the data transmission process and the integrity of the entire project system as a final simulation.

1. Conclusion

Wireless communication systems based on FPGA cards are highly beneficial for industries like defense, medical, automotive, and others due to their reliability, flexibility, and convenience.

The project aims to create a wireless communication system using FPGA cards as its foundation. It involves designing the system architecture and developing communication protocols.

The system outlined in this book enables wireless communication between two FPGA cards, referred to as Card A and Card B. The computer running an application that allows a user to control data transmission. The computer sends data to Card A, which then communicates with Card B wirelessly. Card B interprets the received data and adjusts the intensity of LEDs accordingly. This setup enables remote control of LED intensity through the computer application. The system offers two configurations for wireless communication: a short-range setup using STX882 transmitter and SRX882 receiver with BiPhase coding, and a long-range configuration employing CC1101 with SPI coding. While short-range transmission is simpler, long-range communication is more complex due to module configuration. To enhance the reliability of information transfer, noise reduction filters were implemented along with Debouncer and CRC-8 error detection. Real-time data transmission was achieved by utilizing MAX 10 cards, which increased performance speed. These measures collectively improved the system's reliability and ensured timely data transfer.

The testing process for the system is comprehensive and straightforward, allowing thorough examination of all its components. Test results closely match between the theoretical calculations to the information transfer times and synchronization between units, indicating the system's reliability. However, a notable drawback arises when testing signals with low frequencies, leading to longer simulation times. To mitigate this issue, it's necessary to increase frequencies solely for simulation tests or utilize more powerful computers for faster information processing. This adjustment ensures efficient testing.

During field tests, it was observed that the transmission speed decreased as the distance between the transmitter and receiver increased. For short-range communication, the system was tested up to 60 meters, beyond which the information could still be received but with considerable delay.

In future iterations of the system, incorporating an encryption unit will be a crucial enhancement. This encryption unit will serve to encrypt the information transmitted through the system, ensuring its confidentiality and integrity. This addition is particularly essential to meet the stringent security requirements of customers in industries such as defense and medical, where sensitive information is frequently transmitted.

Implementing a wireless communication system using FPGAs offers significant advantages. Unlike traditional analog-based systems, FPGA-based solutions reduce the need for numerous analog components, minimizing potential errors and streamlining information traffic within the system. Additionally, the entire circuit operates on a pair of FPGA cards, eliminating the complexities associated with transferring wires and constructing intricate circuits. This streamlined approach not only enhances convenience but also simplifies the system's design and implementation process. Overall, leveraging FPGAs for wireless communication systems provides a robust and efficient solution with reduced complexity and improved reliability.

להוסיף את התוצאות של שידור לתווך ארוך

1. References

[1] S. K. Das, *Mobile handset design*. Singapore ; Hoboken, NJ: John Wiley & Sons (Asia), 2010.

[2] F. Qian, Y. Ye, Y. Zhuang, and Y. Shi, “Research on Security Threats and Protection Strategies of Optical Fiber Communication,” in *2019 International Conference on Robots & Intelligent System (ICRIS)*, Haikou, China: IEEE, Jun. 2019, pp. 168–170. doi: 10.1109/ICRIS.2019.00051.

[3] “MIXED SIGNAL VLSI WIRELESS DESIGN”.

[4] K. Iniewski, C. McCrosky, and D. Minoli, *Network Infrastructure and Architecture: Designing High‐Availability Networks*, 1st ed. Wiley, 2008. doi: 10.1002/9780470253526.

[5] “VLSI\_Implementation\_of\_8-PSK\_8-QAM\_16-QAM\_Transmitter\_Circuit\_in\_CMOS\_Technology”.

[6] A. Verma and R. Shrestha, “A New VLSI Architecture of Next-Generation QC-LDPC Decoder for 5G New-Radio Wireless-Communication Standard,” in *2020 IEEE International Symposium on Circuits and Systems (ISCAS)*, Seville, Spain: IEEE, Oct. 2020, pp. 1–5. doi: 10.1109/ISCAS45731.2020.9181188.

[7] A. Chakraborty and A. Banerjee, “A Multiplierless VLSI Architecture of QR Decomposition Based 2D Wiener Filter for 1D/2D Signal Processing With High Accuracy,” in *2018 4th International Conference on Computing Communication and Automation (ICCCA)*, Greater Noida, India: IEEE, Dec. 2018, pp. 1–6. doi: 10.1109/CCAA.2018.8777458.

[8] A. Pang and P. Membrey, “What Is an FPGA and What Can It Do?,” in *Beginning FPGA: Programming Metal*, Berkeley, CA: Apress, 2017, pp. 3–12. doi: 10.1007/978-1-4302-6248-0\_1.

[9] L. Guan, *FPGA-based Digital Convolution for Wireless Applications*. in Springer Series in Wireless Technology. Cham: Springer International Publishing, 2017. doi: 10.1007/978-3-319-52000-1.

[10] R. Ravasz, A. Hudec, D. Maljar, R. Ondica, and V. Stopjakova, “Introduction to Teaching the Digital Electronics Design using FPGA,” in *2022 20th International Conference on Emerging eLearning Technologies and Applications (ICETA)*, Stary Smokovec, Slovakia: IEEE, Oct. 2022, pp. 549–554. doi: 10.1109/ICETA57911.2022.9974732.

[11] E. Monmasson and M. N. Cirstea, “FPGA Design Methodology for Industrial Control Systems—A Review,” *IEEE Trans. Ind. Electron.*, vol. 54, no. 4, pp. 1824–1842, Aug. 2007, doi: 10.1109/TIE.2007.898281.

[12] D. Sundararajan, “Introduction,” in *Control Systems*, Cham: Springer International Publishing, 2022, pp. 1–14. doi: 10.1007/978-3-030-98445-8\_1.

[13] J. Liang and R. Duan, “Design of the Embedded Serial and Parallel Communication Protocol Controller,” in *2010 First International Conference on Pervasive Computing, Signal Processing and Applications*, Harbin, China: IEEE, Sep. 2010, pp. 436–439. doi: 10.1109/PCSPA.2010.111.

[14] J. A. Dell, *Digital Interface Design and Application*, 1st ed. Wiley, 2015. doi: 10.1002/9781119107507.

[15] P. Sharma, A. Kumar, and N. Kumar, “Analysis of UART Communication Protocol,” in *2022 International Conference on Edge Computing and Applications (ICECAA)*, Tamilnadu, India: IEEE, Oct. 2022, pp. 323–328. doi: 10.1109/ICECAA55415.2022.9936199.

[16] A. Loan, “Line Coding,” in *Handbook of Computer Networks*, 1st ed., H. Bidgoli, Ed., Wiley, 2007, pp. 522–537. doi: 10.1002/9781118256053.ch34.

[17] Md. M. Arifin, Md. T. Hasan, Md. T. Islam, Md. A. Hasan, and H. S. Mondal, “Design and Implementation of High Performance Parallel CRC Architecture for Advanced Data Communication,” in *2019 4th International Conference on Electrical Information and Communication Technology (EICT)*, Khulna, Bangladesh: IEEE, Dec. 2019, pp. 1–5. doi: 10.1109/EICT48899.2019.9068750.

[18] N. N. Qaqos, “Optimized FPGA Implementation of the CRC Using Parallel Pipelining Architecture,” in *2019 International Conference on Advanced Science and Engineering (ICOASE)*, Zakho - Duhok, Iraq: IEEE, Apr. 2019, pp. 46–51. doi: 10.1109/ICOASE.2019.8723800.

1. Appendices

Radio waves - are part of the electromagnetic spectrum with a frequency range extending from a few KHz’s to 100’s of GHz, which corresponds to a wavelength in the range to 3 mm. Radio waves are used as a means of conveying information between the transmitter and the receiver of a wireless communications system.