



המכללה האקדמית להנדסה סמי שמעון

המחלקה להנדסת חשמל ואלקטרוניקה

**פיתוח מערכת תקשורת בין מכשירים אלחוטית מבוססת VLSI**  
**Development of a VLSI-Based Wireless Communication System**  
**Between Devices**

פרויקט הנדסי

דו"ח מסכם

הוכן לשם השלמת הדרישות לקבלת  
תואר ראשון בהנדסה B. Sc

מאת

לידור מזרחי  
עמרי אדלן

בהנחיית

מ"ר רמי ברונשטיין

הוגש למחלקה להנדסת חשמל ואלקטרוניקה  
המכללה האקדמית להנדסה אשדוד

06.08.2024

ב' באב ה'תשפ"ד

# **פיתוח מערכת תקשורת בין מכשירים אלחוטית מבוססת VLSI** **Development of a VLSI-Based Wireless Communication System** **Between Devices**

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

תאריך: 06.08.24

תאריך: 06.08.24

תאריך: \_\_\_\_\_

חתימת הסטודנט: 

חתימת המנחה: 

אישור ועדת הפרויקטים: \_\_\_\_\_



המכללה האקדמית להנדסה סמי שמעון

המחלקה להנדסת חשמל ואלקטרוניקה

**תחום הפרויקט: תקשורת**

**סוג הפרויקט: מעשי**

**Keywords: Control Systems, Communications**

**מילות מפתח: מערכות בקרה, תקשורת**

## תקציר

מערכות שליטה מרחוק ממלאות תפקיד בתעשיות שונות כגון צבא ותחבורה על ידי מתן שליטה ממקומות מרוחקים למשל שליטה ברחפנים או רובוטים. 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. כחלק מסקירת הספרות, ניתנים הסברים מפורטים לגבי הידע הדרוש לתכנון ובניית המערכת. על מנת לפתח מערכת כזו יש צורך להשתמש בשני בקרי FPGA והתקנים חיצוניים שיפורטו בהמשך. יתרה מזאת, תוצג סכמה המציגה את אופן מימוש המערכת תוך מתן הסבר על דרך פעולתה. סכמה זו מבהירה את הפונקציונליות של כל רכיב, את תפקודה הכולל של המערכת ואת השיקולים האלגוריתמיים המאפשרים לה לעמוד בדרישות שצוינו. החלק הסופי דן בבדיקות נפרדות וסימולציות שנערכו על רכיבי מערכת בודדים והערכה מקיפה של תקינות המערכת כולה באמצעות ModelSim, תוך מתן דגש על זמני מעבר המידע. פרויקט זה נועד להציג את היישום המעשי של מערכות שליטה מרחוק המשתמשות בטכנולוגיית FPGA, תוך עמידה בקריטריונים ביצועיים ותרחישי העולם האמיתי.

## Abstract

Remote control systems fulfill roles in various industries such as military and transportation by providing control from distant locations such as drones and robotics. 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. As a part of literature review, 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. Furthermore, a scheme will be presented showing how the system is implemented while explaining how it works. This scheme clarifies the functionality of each component, the overall functioning of the system and the algorithmic considerations that allow it 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.

## Contacts

<b>1. Introduction</b>	<b>1</b>
1.1 Internet of Things	1
1.2 Challenges and Solutions for Managing Robots and Drones in Hazardous Environments	1
1.2.1 Wireless Communication and Wireless Communication System	2
1.2.2 Privacy Protection	2
1.3 Project Description and Requirements	3
<b>2. Literature Review</b>	<b>4</b>
2.1 System Overview	4
2.2 VLSI Technology	5
2.3 FPGA	5
2.4 Control System	7
2.5 UART Communication	7
2.6 Serial and Parallel Communication	7
2.7 BiPhase	8
2.8 CRC	9
<b>3. System Structure</b>	<b>10</b>
3.1 Laptop Computer	10
3.1.1 Python	10
3.1.1.1 Serial Configuration	10
3.1.1.2 GUI Initialization	10
3.1.1.3 Event Handling Functions	11
3.1.1.4 CRC Calculation Function	11
3.1.1.5 Main Loop	12
3.1.1.6 Packet Array	12
3.1.2 USB transformation TTL adapter	13
3.2 Card A- short range transmission	14
3.2.1 STX882:	15
3.2.2 Block 1: "Uart_tx_Constant"	16
3.2.3 Baud Clock Generation Process ("baud_clk")	16

3.2.4 Rising Edge Detection Process ("rising_edg")	17
3.2.5 UART Transmission Finite State Machine Process ("transmission")	17
3.2.6 Block 2: "Uart_rx"	19
3.2.7 Block 3: "Ram2_x"	23
3.2.8 Block 4: "BiPhase_tx"	24
3.2.9 Block 5: "Card_A_Design_Python"	27
3.3 Card B - short range transmission	28
3.3.1 SRX882:	29
3.3.2 Block 1: "BS_Filter"	30
3.3.3 Block 2: "Simple_BS"	31
3.3.4 Block 3: "CRC8BIT"	32
3.3.5 Block 4: "Data_Organizer"	34
3.3.6 Block 5: "RGB"	36
3.3.7 Block 6: "Card_B_Design"	39
3.4 SPI - long range transmission	40
3.4.1 spi_cc1101_write	42
3.4.2 spi_cc1101_read	43
4. A Set of Final Tests	44
4.1 Test Bench 1: "Uart_tx_Rom"	45
4.2 Test Bench 2: "Uart_rx"	47
4.3 Test Bench 3: "Card_A_Design"	48
4.4 Test Bench 4: "Test_BS"	49
4.5 Test Bench 5: "Test_CRC_DO"	51
4.6 Test Bench 6: "Card_B_Design"	52
4.7 Logic Analyzer: "spi_cc1101_read/write" check	53
5. Conclusion	54
6. References	55
7. Appendices	57
7.1 Python Interface Code:	57
7.2 Tables	63

## List of Figures

Figure 1.2 Basic block diagram of communication system .....	2
Figure 2.1 General view of system block diagram .....	4
Figure 2.2 General architecture of FPGA [13] .....	5
Figure 2.3 General architecture of CLB [13] .....	6
Figure 2.4 Biphas waveforms- biphas-S [19] .....	8
Figure 3.1 Python Graphical Control Interface .....	11
Figure 3.2 Update Array Main Loop Every 30 Milliseconds .....	12
Figure 3.3 Packet Array .....	12
Figure 3.4 USB transformation TTL adapter: SH-U09C2 .....	13
Figure 3.5 Card A short range block diagram .....	14
Figure 3.6 STX882 transmitter .....	15
Figure 3.7 Rising & Falling Edge Detection .....	17
Figure 3.8 FSM transitional .....	17
Figure 3.9 Right Shift Register on The Packet .....	18
Figure 3.10 Sampling Incoming Data Bits .....	19
Figure 3.11 Signal Before and After Debouncer .....	20
Figure 3.12 Debouncer RTL .....	20
Figure 3.13 UART Byte Receiver .....	21
Figure 3.14 "Uart_rx" Algorithm Schematic .....	22
Figure 3.15 Scheme Ram Block .....	23
Figure 3.16 BiPhase Transmitter Main FSM .....	25
Figure 3.17 BiPhase Transmitter Side FSM .....	26
Figure 3.18 BiPhase Encoder Wave Format .....	26
Figure 3.19 Connection Diagram "Card_A_Design_Python" .....	27
Figure 3.20 Card B short range block diagram .....	28
Figure 3.21 SRX882 receiver .....	29
Figure 3.22 Filtering Signal .....	30
Figure 3.23 BiPhase Decoder Wave Format .....	31
Figure 3.24 CRC-8 RTL Algorithm Transitional Diagram .....	32
Figure 3.25 CRC-8 Check Operation Flow Chart .....	33
Figure 3.26 RGB Frame Composition .....	34
Figure 3.27 Data Organizer Operation .....	35
Figure 3.28 Code Timing .....	36
Figure 3.29 Sequence Chart .....	36
Figure 3.30 Data Transmission Method .....	37



Figure 3.31 LEDs Management .....	38
Figure 3.32 Connection Diagram "Card_B_Design" .....	39
Figure 3.33 Card A long range block diagram .....	40
Figure 3.34 Card B long range block diagram .....	40
Figure 3.35 Configuration Registers Write and Read Operations .....	41
Figure 3.36 SPI Write Module FSM .....	42
Figure 3.37 SPI Write Module FSM .....	43
Figure 4.1 Uart_tx simulation start .....	45
Figure 4.2 "Uart_tx" Simulation data transfer .....	46
Figure 4.3 Rom mif file.....	46
Figure 4.4 "Uart_rx" Simulation data transfer .....	47
Figure 4.5 "Card_A_Design" Simulation data transfer to "BiPhase_tx_out" .....	48
Figure 4.6 "BS_Filter" & "Simple_BS" Simulation sync "nrzl_data" & "main_clk" .....	49
Figure 4.7 "BS_Filter" & "Simple_BS" Simulation data transfer to "nrzl_data" .....	50
Figure 4.8 "CRC8BIT"&"Data_Organizer" Simulation CRC Check & Data Organize .....	51
Figure 4.9 "Card_B_Design" Simulation, The System Output.....	52
Figure 4.10 Writing and Reading Process Shown in Logic Analyzer .....	53
Figure 4.11 Reading from CC1101 .....	53

## List of Tables

Table 3.1 "Uart_tx_Constant" Ports .....	63
Table 3.2 "Uart_tx_Constant" Internal Signals .....	63
Table 3.3 "Uart_rx" Ports .....	64
Table 3.4 "Uart_rx" Internal Signals .....	64
Table 3.5 "Ram2_X" Ports.....	65
Table 3.6 "BiPhase_tx" Ports.....	65
Table 3.7 "BiPhase_tx" Internal Signals.....	66
Table 3.8 "BS_Filter" Ports .....	67
Table 3.9 "BS_Filter" Internal Signals .....	67
Table 3.10 "Simple_BS" Ports.....	67
Table 3.11 "Simple_BS" Internal Signals .....	68
Table 3.12 "CRC8BIT" Ports.....	69
Table 3.13 "CRC8BIT" Internal Signals .....	69
Table 3.14 "Data_Organizer" Ports .....	70
Table 3.15 "Data_Organizer" Internal Signals.....	70
Table 3.16 "RGB" Ports.....	71
Table 3.17 "RGB" Ports Internal Signals.....	71

## **1. Introduction**

This chapter provides background information about our project. We will describe the challenges of IoT systems, and the goals of the system as dictated by its requirements.

### **1.1 Internet of Things**

Internet of Things (IoT) involves devices that interact with their environment mechanically or electrically while transmitting data. Driven by hardware innovations like microcontrollers and field-programmable gate arrays (FPGAs), IoT has transformed device interaction and communication by connecting physical objects embedded with electronics, software, and communication capabilities. This integration between the physical and digital worlds allows these objects to collect, exchange, and remotely control data.

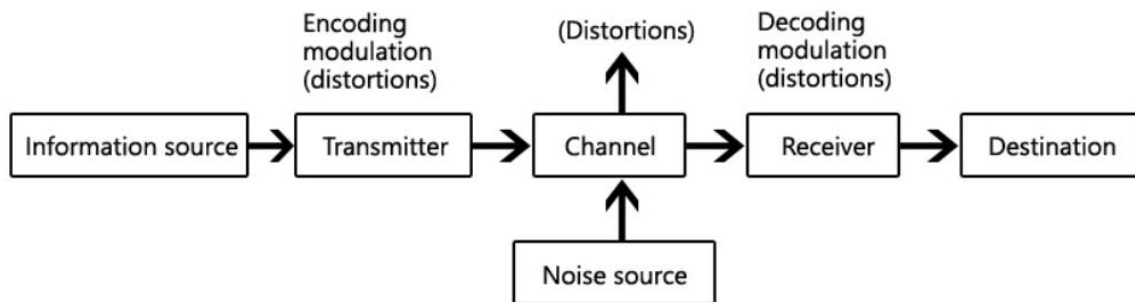
IoT technology enhances drones and robotics by enabling real-time data collection and analysis. Drones use IoT to perform tasks like aerial surveillance, environmental monitoring, and delivery services, utilizing sensors and cameras to make immediate decisions. In robotics, IoT allows systems to monitor performance, predict maintenance needs, and adjust operations in real-time, boosting efficiency and safety [1], [2]. Central to the functionality of IoT devices is wireless communication, which provides the necessary connectivity for data transmission and remote control.

### **1.2 Challenges and Solutions for Managing Robots and Drones in Hazardous Environments**

Deploying robots or drones in dangerous environments, such as mining areas, urban disasters, hostage situations, explosions, and more, poses significant management challenges in terms of communication and safety. Ensuring robust communication channels between operators and robots is essential for maintaining control and relaying data in real-time. Safety protocols must be designed to prevent malfunctions that could endanger both robots and humans. Additionally, equipping robots with advanced sensors enhances their effectiveness in these conditions [3]. One solution to avoid endangering humans and to control the robots remotely is the remote-control device, an interface used by the operator to send commands to the vehicle or robot, which can be a physical remote control with joysticks, buttons, and other input methods.

## 1.2.1 Wireless Communication and Wireless Communication System

Effective approach to enhance communication and control in such scenarios is the implementation of wireless communication systems. Wireless communication transmits information between points using electromagnetic waves, particularly radio waves, without a continuous guided medium. A simple wireless communication system includes key components working together: the transmitter generates and modifies the information signal (voice, data, etc.) for wireless transmission, converting it into electromagnetic waves. The channel, which is the air or free space, carries these waves to the receiver. The receiver, captures the waves, extracts and enhances the original signal, filters out noise and interference, decodes the information, and delivers it to the end user [4], [5]. An example illustrating the simple wireless communication system is shown in Figure 1.1:



**Figure 1.1 Basic block diagram of communication system**

However, wireless communication faces several challenges. Interference from other electronic devices, physical obstacles, and environmental factors can degrade signal quality. Additionally, the limited spectrum and bandwidth constraints can impact the efficiency of wireless networks. Addressing these challenges is essential for optimizing wireless communication systems.

## 1.2.2 Privacy Protection

One of the challenges in security application is ensuring security during the transmission of data over the air. This vulnerability can lead to exposing sensitive information such as secret medical documents or classified military data. Additionally, personal and financial details like credit card information are at risk. To combat these security issues, industry solutions include the development of encoders and encryption algorithms designed for IoT systems. These technologies encrypt data during transmission, making it more difficult for unauthorized parties to access and misuse the information. Implementing those encryption methods is essential for protecting data and confidentiality in wireless communication systems [6].

### **1.3 Project Description and Requirements**

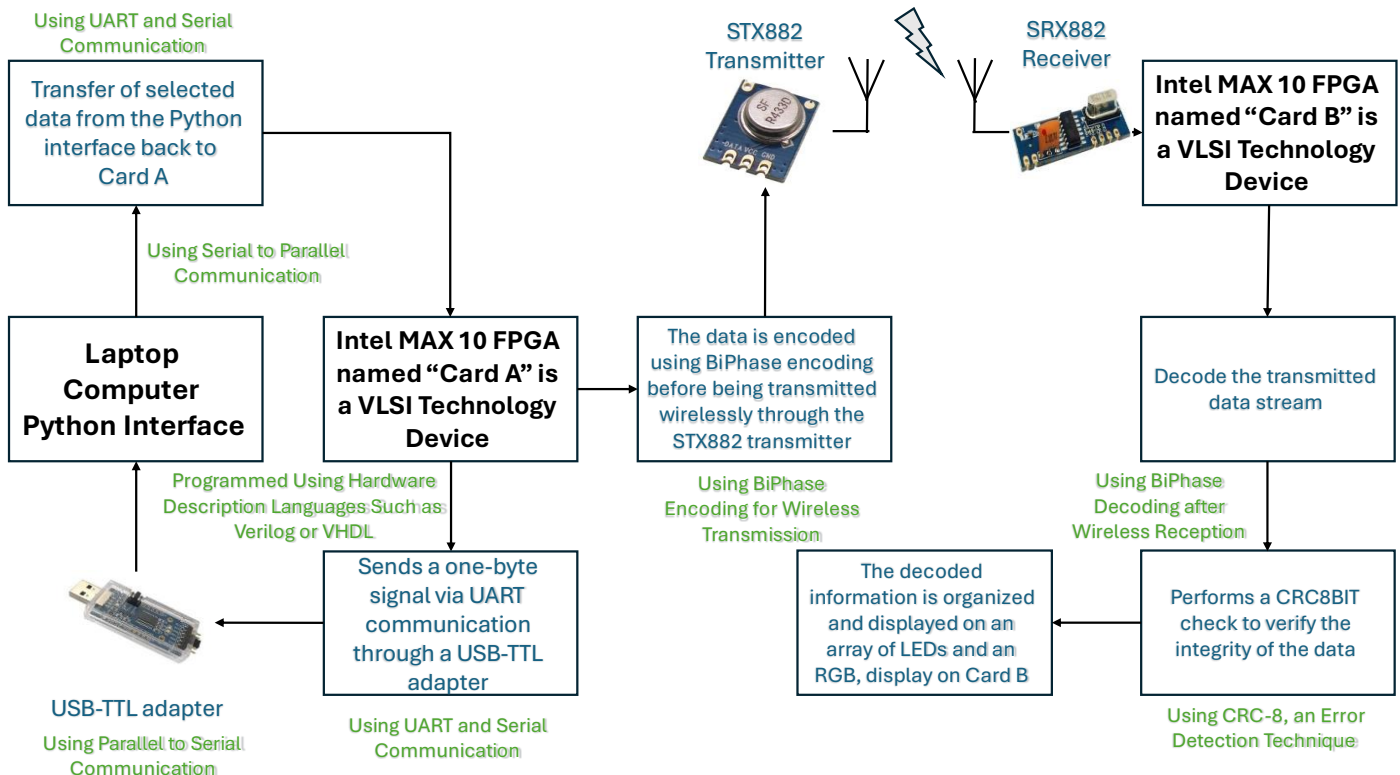
This project aims to develop a secure, reliable, and efficient remote-control system for vehicles, robots or drones operating in hazardous environments. We created a wireless system where a PC serves as the control station, using a Python program to simulate analog control voltages represented by RGB colors on the display. These colors indicate different analog commands and their intensities, while three on-screen buttons simulate discrete commands. The system utilizes two FPGA MAX 10 cards: one connected to the PC via a USB TTL adapter, which transmits commands using the STX882 transmitter for short-range and the CC1101 transceiver for long-range communication. At the remote site, the second FPGA card, equipped with the SRX882 receiver, receives and decodes the data into control commands, which are displayed using LEDs and RGB LEDs to provide visual feedback.

The project requirements include establishing a wired interface between the PC and FPGA, implementing wireless communication between the two FPGAs, encrypting the information transmitted wirelessly, and incorporating error-checking mechanisms such as CRC. The system supports both short and long-range wireless transmission and ensures real-time operation with minimal latency. Additionally, a mobile interface, either computer-based or app-based, is developed for remote control. In the next chapter, we will explore the tools needed to understand this system, including FPGA controllers, serial and parallel communication protocols, signal decoding/encoding, and error checking methods.

## 2. Literature Review

### 2.1 System Overview

The project is a wireless data transmission system utilizing STX882 modules to transfer data from a computer to a Max 10 FPGA board. Data is encoded, transmitted wirelessly, and then decoded and displayed on the FPGA board, with error checking implemented for data integrity. The structure of wireless communication remote-control system with the two FPGA cards is shown in Figure 2.1:



**Figure 2.1 General view of system block diagram**

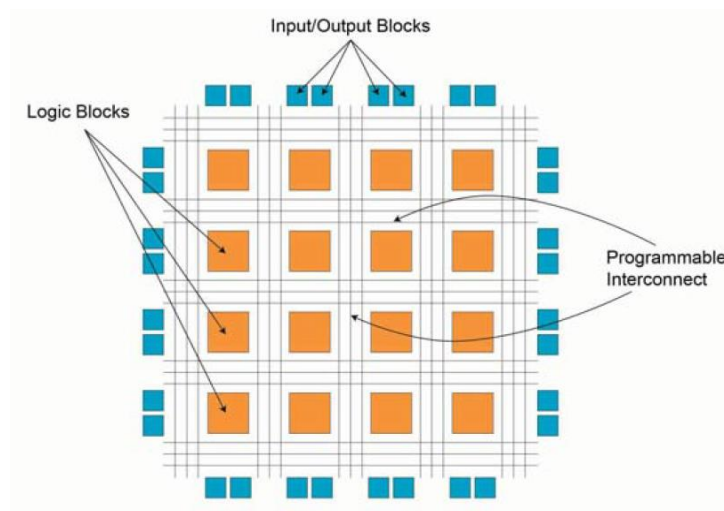
To achieve such complex functionality and integration, VLSI technology plays a role in the design and implementation of the system's hardware components.

## 2.2 VLSI Technology

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, allowing for compact, energy efficient and high-performance implementation. 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 [7]. In the context of wireless communication system, VLSI chips can be dedicated to handling tasks such as modulation, demodulation, encoding, decoding and signal processing [8], [9], [10].

## 2.3 FPGA

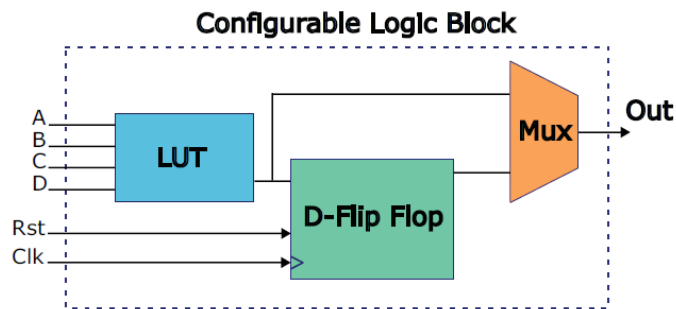
Field programmable gate array (FPGA) is a type of digital integrated circuit chip that provides reconfigurable parallel processing capability. This feature allows multiple machines to work together seamlessly in real time and, can be programmed using hardware description languages such as Verilog or VHDL, and has three basic building blocks: logic gates, flip-flops + memories, and wires [11], [12]. 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 2.2:



**Figure 2.2 General architecture of FPGA [13]**

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 2.3:



**Figure 2.3 General architecture of CLB [13]**

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.

FPGAs come in several types, each with distinct characteristics. SRAM-based FPGAs use volatile memory, requiring configuration data to be reloaded on each power-up, but offer high performance and flexibility. Anti-Fuse FPGAs use a one-time programmable technology, creating permanent connections during programming. EEPROM-based FPGAs, similar to flash-based ones, use non-volatile memory, but typically have slower write speeds. Flash-based FPGAs use non-volatile memory, retaining their configuration when powered off.

In this project, we used an Intel MAX 10 FPGA, which is designed with embedded flash memory which allows for quick startup and is energy-efficient [13], [14].



## 2.4 Control System

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 [15].

## 2.5 UART Communication

Universal Asynchronous Receiver Transmitters (UARTs) facilitate parallel data transmission over a serial line, commonly using the RS-232 standard for serial communication between computers and peripherals. Voltage converter chips bridge the voltage gap between RS-232 and FPGA I/O pins. A UART includes a transmitter that converts parallel data to serial format and a receiver that does the reverse [16].

Data transmission begins with a start bit (logical '0') to synchronize the receiver's clock, followed by data bits, an optional parity bit for error detection, and stop bits (logical '1') to mark the frame's end. The baud rate, data bits, and stop bits must be pre-agreed by both the transmitter and receiver. Common baud rates range from 2400 to 115200 bauds.

To accurately retrieve data, the receiver uses oversampling to estimate bit middle points, forming the basis for clock signal generation. The UART receiver subsystem uses oversampling schemes of 16, 24, and 32 for 1, 1.5, and 2 stop bits, respectively [17].

## 2.6 Serial and Parallel Communication

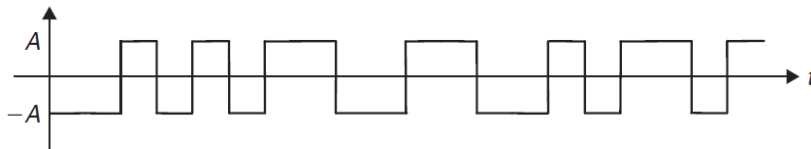
Serial communication, facilitated by Universal Asynchronous Receiver-Transmitter (UART) controllers, enables bidirectional data exchange between the Systems-on-Chip and external peripherals. These controllers support full-duplex communication, allowing simultaneous transmission and reception of data. 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 [18]. When transmitting serial data wirelessly, maintaining data integrity and synchronization is essential, and this is achieved through biphasic encoding techniques.

## 2.7 BiPhase

Manchester encoding is a prominent biphas coding scheme that ensures reliable clock recovery at the receiver's end. It guarantees a transition in the middle of each bit period, allowing the receiver to accurately extract the clock signal from the data stream, maintaining synchronization between the transmitter and receiver.

Biphase-S encodes binary data by making a transition at the start of each bit. For a binary 0, a second transition occurs in the middle of the bit interval, while for a binary 1, there is no second transition. This method enables synchronization and clock recovery by including timing information within each bit interval, eliminating DC wander.

Biphase-mark code introduces transitions at the start of each bit interval and an additional transition at the midpoint for binary 1s, aiding in timing synchronization. Differential Manchester encoding improves reliability by using transitions to represent binary values, providing immunity to polarity reversals and enhancing error detection. Hybrid schemes like Code Mark Inversion (CMI) and Differential Mode Inversion (DMI) combine biphas coding with other techniques for energy-efficient transmission in diverse environments [19]. An example of Biphase waveforms - biphase-S is shown in Figure 2.4:



**Figure 2.4 Biphase waveforms- biphase-S [19]**

## 2.8 CRC

Cyclic Redundancy Check (CRC) codes use Linear Feedback Shift Registers (LFSRs) to create a signature for detecting data corruption. Serial LFSRs produce one bit of output per clock cycle, which suits high-speed applications.

CRC-8 is an error-detection method that generates an 8-bit checksum from the data. It starts with an initial value (usually zero) and processes each bit by XORing the CRC value with the data bit, shifting right, and XORing with a predefined polynomial if needed. After processing all bits, the resulting CRC value is compared with the received CRC. A match indicates correct data, while a mismatch suggests an error. CRC-8 detects errors but cannot correct them and its effectiveness depends on the polynomial used.

There are several CRC polynomials, notable examples include:

$$CRC8: P(x) = x^8 + x^2 + x^1 + 1 \text{ (HEX - 07)} \quad (2.1)$$

$$CRC12: P(x) = x^{12} + x^{11} + x^3 + x^2 + x^1 + 1 \text{ (HEX - 80F)} \quad (2.2)$$

$$CRC16: P(x) = x^{16} + x^{15} + x^2 + 1 \text{ (HEX - 8005)} \quad (2.3)$$

$$CRC32: P(x) = x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^8 + x^7 + x^5 + x^4 + x^2 + x^1 + 1 \text{ (HEX - 04C11DB7)} \quad (2.4)$$

These polynomials influence error detection. For many systems, CRC-8 is sufficient if its error rate meets the system's needs. Shift register operations in hardware use these polynomials for CRC computation by shifting and XORing bits [20], [21].

### **3. System Structure**

In this chapter, we will detail the structure of wireless communication remote-control system of the two FPGA cards. Initially we will present the python interface, then we will characterize each unit separately.

#### **3.1 Laptop Computer**

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

##### **3.1.1 Python**

The Python code uses libraries to establish seamless communication between the computer and the microcontroller. Moreover, the user-friendly application interface simplifies the manipulation of RGB and LED lights.

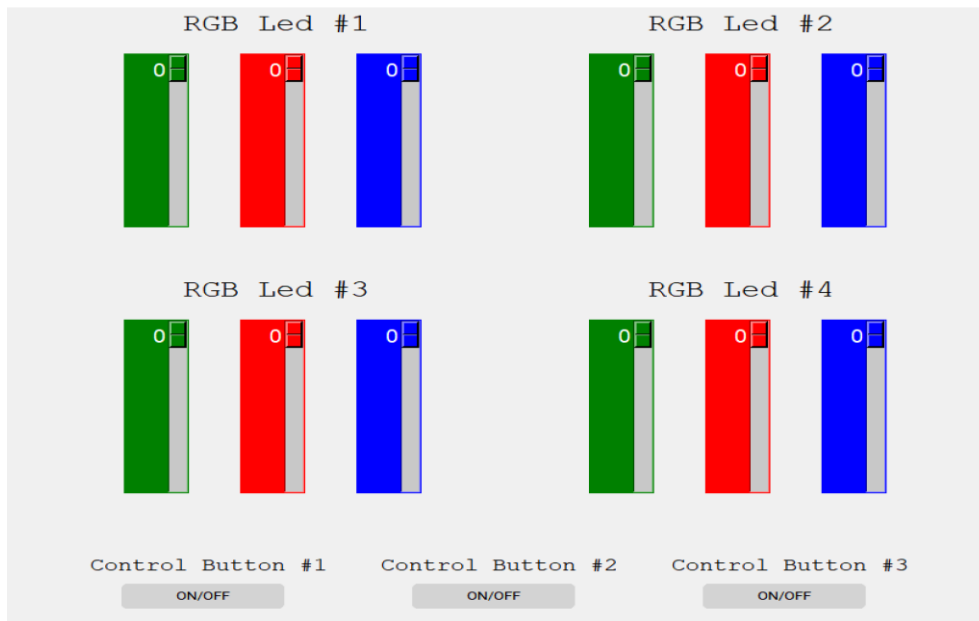
##### **3.1.1.1 Serial Configuration**

The script configures the serial port (COM5) 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.

##### **3.1.1.2 GUI Initialization**

The script initializes a Tkinter window ("root") with dimensions of 850x700, providing a graphical interface for user interaction, is shown in figure 3.1:



**Figure 3.1 Python Graphical Control Interface**

### 3.1.1.3 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.

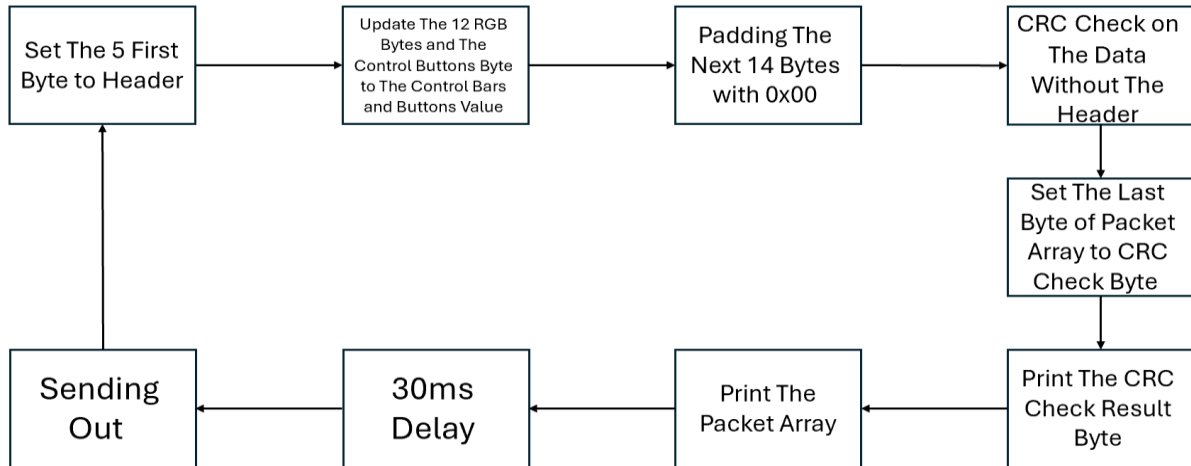
### 3.1.1.4 CRC Calculation Function

"`calculate_crc8()`" is a function that calculates an 8-bit CRC checksum for a given data array using the CRC8 polynomial (0x07) and stores this checksum in the last byte of 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.

### 3.1.1.5 Main Loop

The `root.mainloop()` 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.

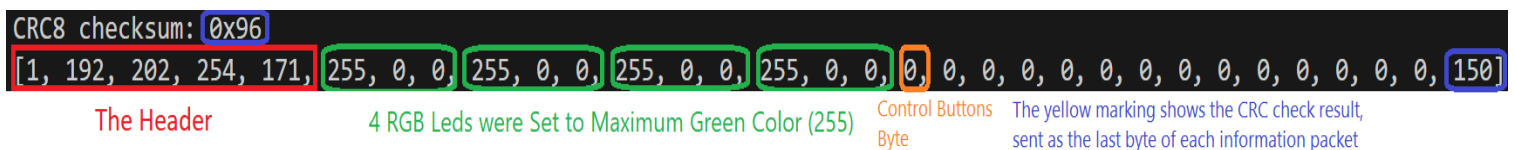
Update array main loop every 30ms is shown in figure 3.2:



### Figure 3.2 Update Array Main Loop Every 30 Milliseconds

#### 3.1.1.6 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. This structured array serves as a comprehensive data packet, combining user inputs, control signals, and error-checking mechanisms. 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. Example to packet array is shown in figure 3.3:



### Figure 3.3 Packet Array

### 3.1.2 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 3.4:

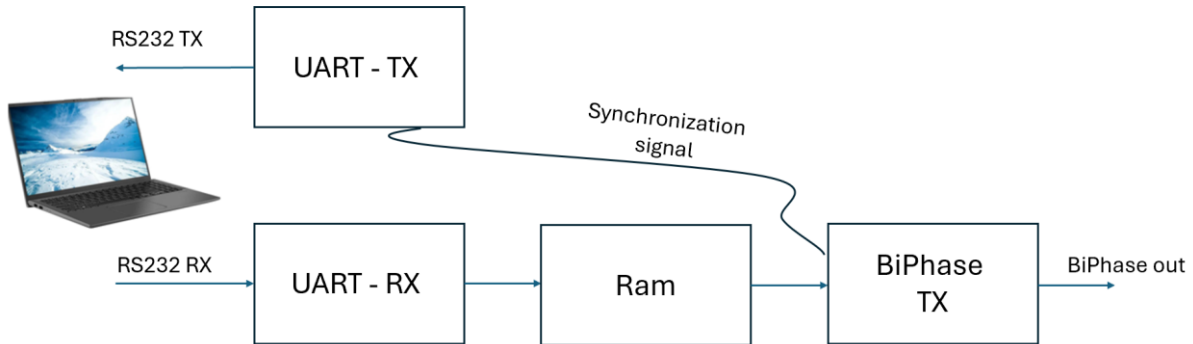


**Figure 3.4 USB transformation TTL adapter: SH-U09C2**

In our project, the USB to TTL adapter serves as an intermediary, enabling 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. 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 data transfer rates of up to 3 Mbps, allows for 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.

### 3.2 Card A- short range transmission



**Figure 3.5 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.

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. The following diagram, shown in Figure 3.5, 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.



### 3.2.1 **STX882:**

The STX882 is a wireless RF transmitter module commonly used in remote control applications. The working frequency of 433MHz, it achieves 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 3.6:



**Figure 3.6 STX882 transmitter**

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.

### 3.2.2 **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.

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.

### 3.2.3 **Baud Clock Generation Process ("baud\_clk")**

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, the baud clock is generated every 651 cycles of the system clock, every 26,040 ns (38,400Hz).

Choosing a UART frequency of 38,400Hz involves several considerations. Initially, the frequency is sufficiently rapid to fill the RAM 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.

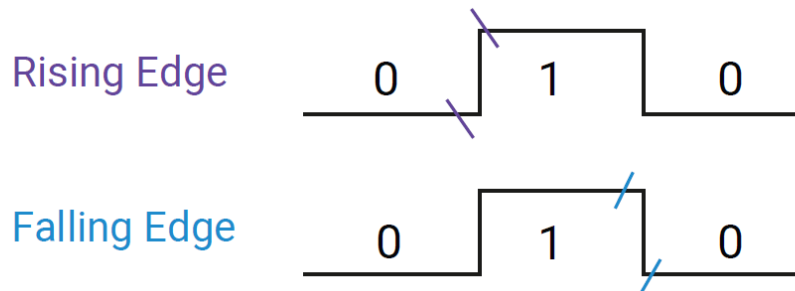
$$\frac{1}{38400[Hz]} (uart\ transmit\ 1\ bit) \times 12(packet) \times 32(bytes) \cong 10 \times 10^{-3} = 10ms \quad (3.1)$$

$$\frac{1}{4800[Hz]} (uart\ transmit\ 1\ bit) \times 12(packet) \times 32(bytes) \cong 80 \times 10^{-3} = 80ms \quad (3.2)$$

An additional factor involves sending 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 maintain proximity to the subsequent transmission. Opting for a frequency of 38,400 Hz aligns with both considerations.

### 3.2.4 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 is synchronization the clock signal. Rising and falling edge detection is shown in figure 3.7:

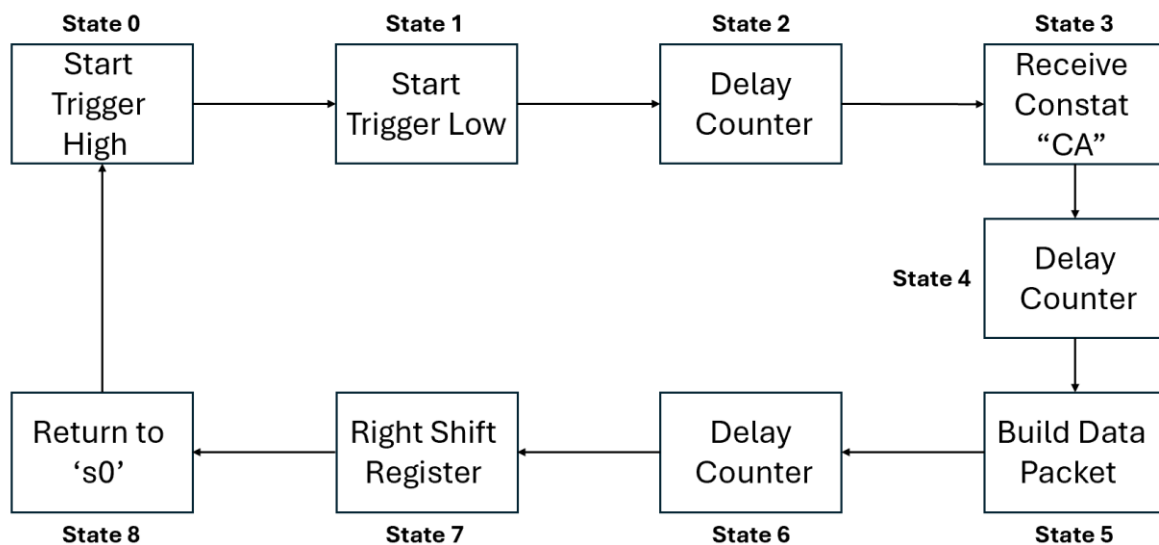


**Figure 3.7 Rising & Falling Edge Detection**

### 3.2.5 UART Transmission Finite State Machine Process ("transmission")

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 trigger pulse signal. The FSM controls the generation of start and stop bits, constructs serial data packets for asynchronous communication and transmit this packet to the computer by right shift register. For system stability, there are delays between the states.

FSM flow chart is shown in Figure 3.8:



**Figure 3.8 FSM transitional**

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 by right shift register.

Example to how to generate a right shift register on the packet is shown in Figure 3.9:



**Figure 3.9 Right Shift Register on The Packet**

## 3.2.6 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 check if 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.

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

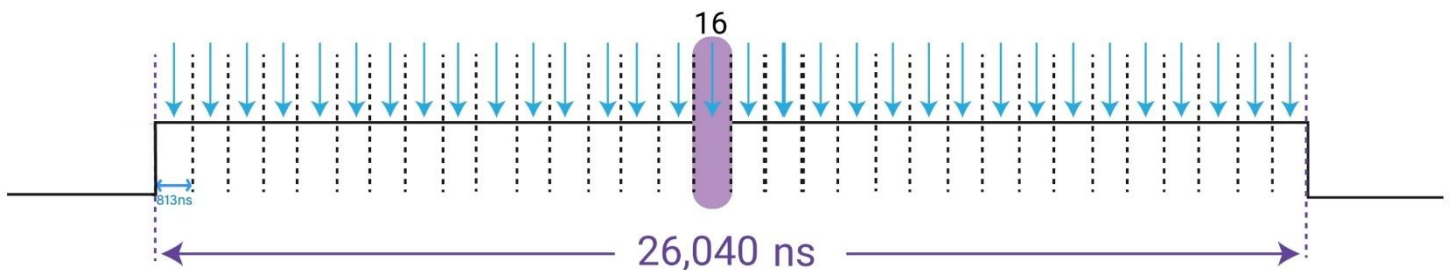
The architecture implements a UART receiver, generate a clock signal with a frequency 32 times higher than the baud clock of the UART communication. This fast clock samples incoming data bits in the middle of each 32-bit division, allowing sampling in a stable area.

Baud rate clock calculation:

$$38,400(\text{baud clock frequency}) \times 32(\text{samples}) = 1,228,800 \text{ Hz} \quad (3.3)$$

$$\frac{1}{1,228,800} \cong 813 \text{ ns} \gg \frac{813}{20 (\text{sysclkperiod})} = 40 \gg \text{when half period is 20 times clocks} \quad (3.4)$$

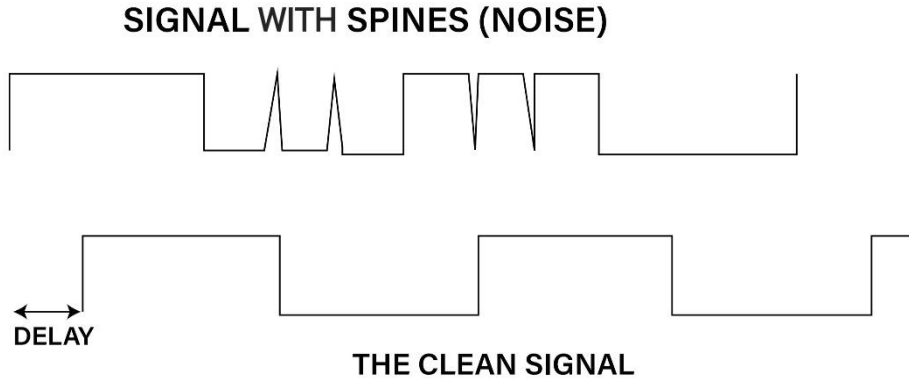
How the sampling incoming data bits work is shown in figure 3.10:



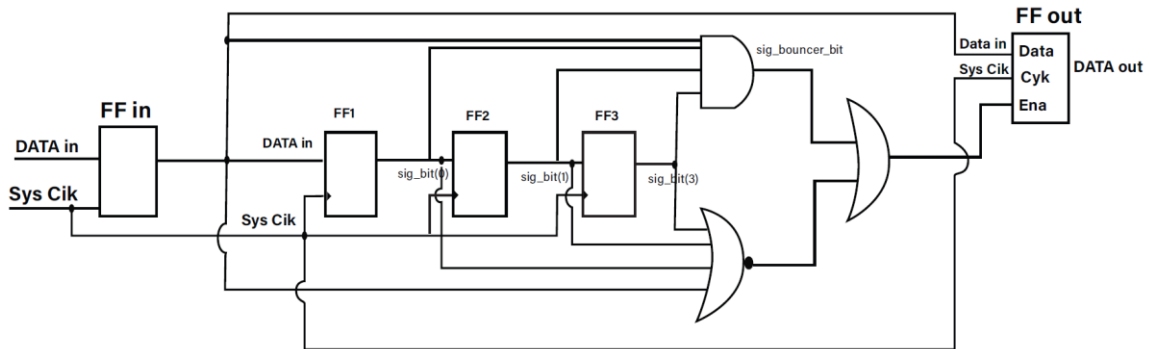
**Figure 3.10 Sampling Incoming Data Bits**

The debouncer operation is performed before sampling the incoming data bits. Debouncing is a technique used to eliminate noise or jitter in a digital signal. The debouncer check three bits and if all three bits are equal, it means that the signal has remained stable for three consecutive clock cycles, in such a case, assigned the value of the stable bit, effectively debouncing the signal.

Example of how debouncer works is shown in Figure 3.11 and the following diagram, shown in Figure 3.12, depicts the debouncer RTL:



**Figure 3.11 Signal Before and After Debouncer**



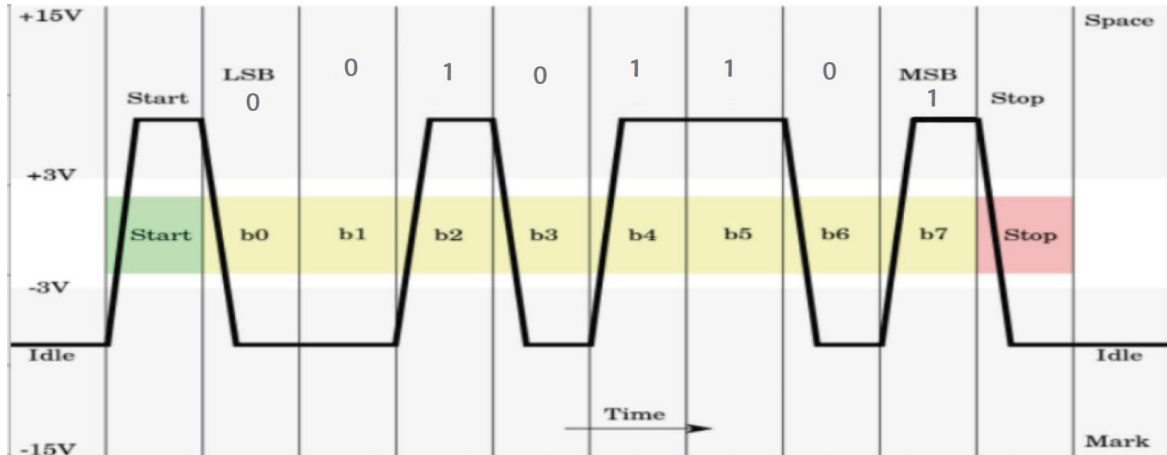
**Figure 3.12 Debouncer RTL**

The reception process implements a finite state machine (FSM) for UART reception. overseeing the reception and processing of incoming UART data. The reset RAM address counter reaches 500,000 (10ms, A delay that ends before the transmission time) when there is no data for 500,000 clock cycles. The chosen delay interval is carefully planned to finish before the current transmission concludes. This strategic timing resets the address before the start of the next transmission, so the RAM is ready to promptly receive the upcoming information well in advance.

To receive the data transmitted from the computer through UART communication, each bit will be sampled in its center with signal with a frequency 32 times higher than the baud clock of the UART communication. For accurate reception of data sent from the computer at a rate of 38,400 Hz, we will focus on the 16th bit. The non-ideal shape of the received bits, as depicted in Fig 3.11 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 3.13:



**Figure 3.13 UART Byte Receiver**

When a start bit '0' is detected, the following 8 bits are identified as data bits, continuing until the end bits are received, and write this detected byte to the current RAM address.

```
sig_ram_address <= not toggle & sig_cnt_address
```

The 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. When one half stores data from the computer, the other half is read by the transmitter in preparation for transmission to card B. After the transmission process is completed, the next set of data is stored in the half of the memory that was used for reading, and the other half is readied for the next transmission and so on. This operation allowing a continuous flow of data receiving and sending without any delay.

The Full FSM flow chart of receiving the information from the computer via UART communication and storing it in the RAM memory is shown in Figure 3.14:

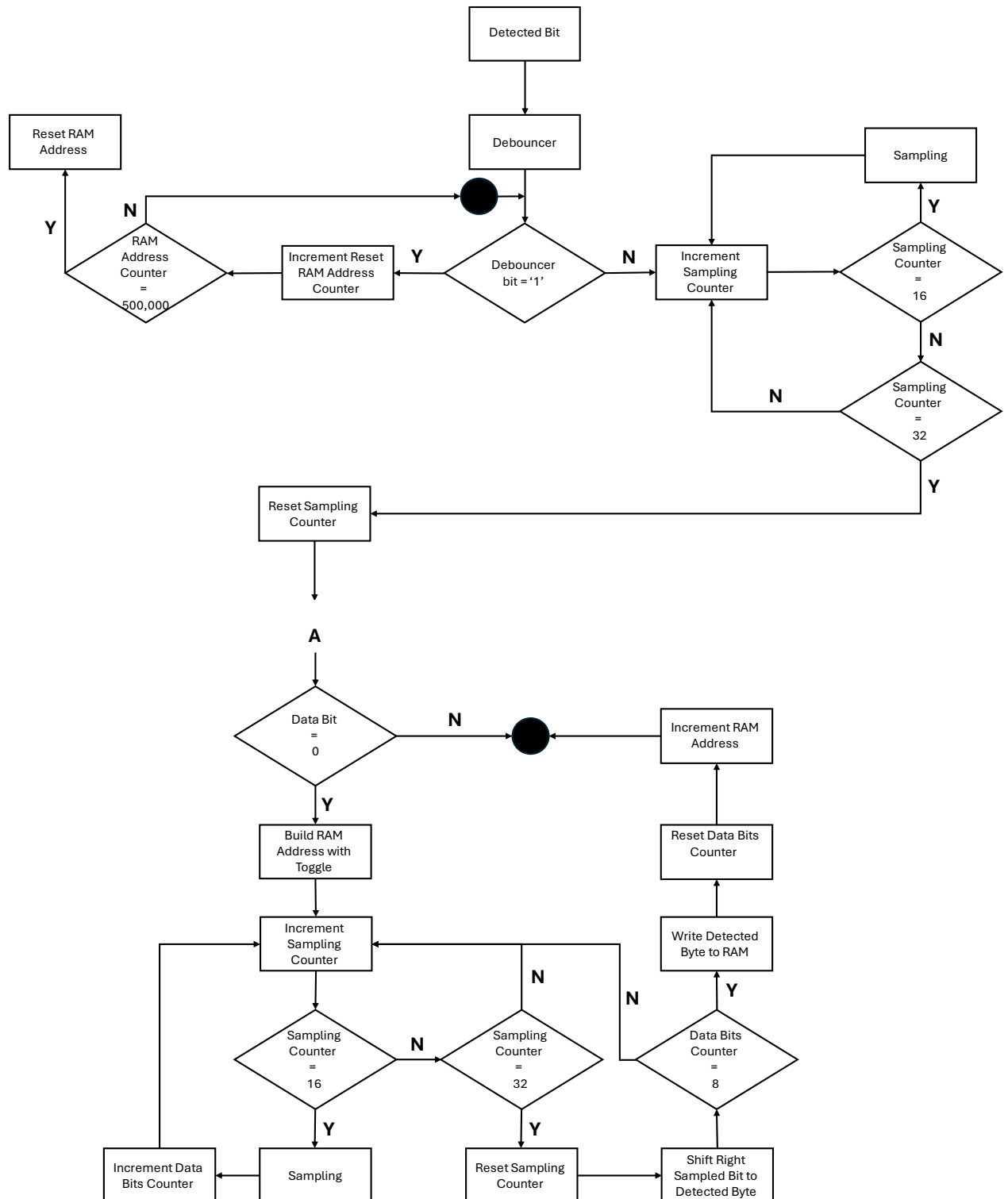
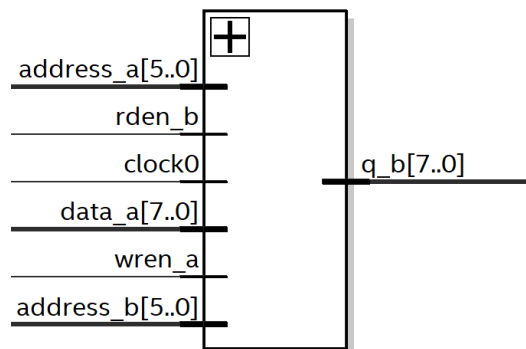


Figure 3.14 "Uart\_rx" Algorithm Schematic



## 3.2.7 Block 3: "Ram2\_x"

RAM (Random Access Memory) contains multiplexing and 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. RAM memory block is shown in Figure 3.15:



**Figure 3.15 Scheme Ram Block**

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" block sends data to the first 32 addresses, while simultaneously, data from the other 32 addresses is sent to "BiPhase" block. 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 stores the data received from "Uart\_rx" block and forwards it to "BiPhase" block during transmission. Without "Ram2\_X," data could be lost because "Uart\_rx" block operates at a much faster rate (38400 Hz), where each bit is sent in 26 microseconds, compared to "BiPhase" block, which takes 328 microseconds to transfer each bit. This speed difference means that "Uart\_rx" block could potentially send around 12 bits before "BiPhase" block manages to receive them, resulting in data loss. Therefore, "Ram2\_X" block acts as a buffer, saving the incoming data until "BiPhase" block is ready to receive it.

### 3.2.8 Block 4: "BiPhase\_tx"

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.

The architecture implements a biphaser encoder, generating a clock signal 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. Every  $2^{13}$  rising edge the clock signal change the logic state, this operation generate the baud rate clock (main clock) of BiPhase.

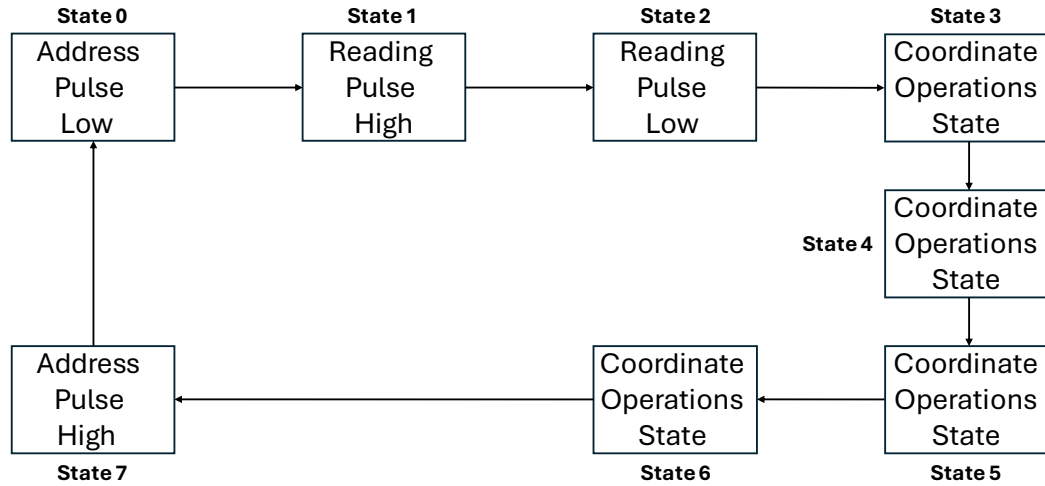
$$2^{13} * 20 * 10^{-9} = 164 \mu s \text{ ( half priod)} \quad (3.5)$$

$$T = 164 * 10^{-6} * 2 = 328 \mu s \quad (3.6)$$

$$F = \frac{1}{T} = \frac{1}{328 * 10^{-6}} = 3048 \text{ Hz} \quad (3.7)$$

Data transfer unfolds through serial communication, organized by a finite state machine (FSM). This main FSM controls the process of reading data from RAM in a synchronized manner, where reading pulse enables data reading and address pulse controls the increment of memory addresses. The FSM cycles through states to coordinate these operations, allowing sequential

data retrieval (using the other timing processes). FSM flow chart is shown in Figure 3.16:



**Figure 3.16 BiPhase Transmitter Main FSM**

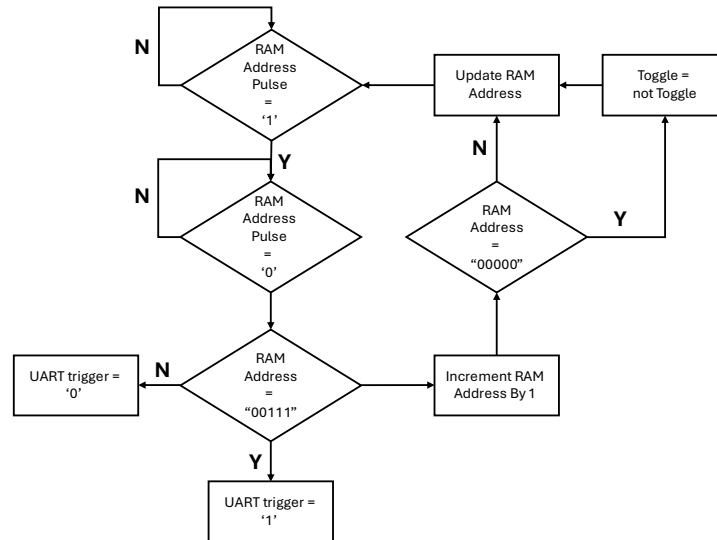
The operation to initiator for UART transmission, effectively signaling the UART transmitter unit to commence data transfer. Aligning the transmission precisely within the system's operational timeline, and precise synchronization. It is possible to send the UART trigger signal up to 28 bytes, which is used as an indicator to update the new data in the RAM, close to the start of the next transmission cycle. This determination is guided by the following calculation: 4 transmitted bytes, 10 milliseconds are required. Loading all 32 bytes into RAM also takes 10 milliseconds.

$$4 \text{ Bytes Transfer} = 4 [\text{Bytes}] * 8 [\text{States}] * 328 * 10^{-6} [\text{Main Clock}] = 21ms \quad (3.8)$$

Additionally, it governs the timing of RAM address progression for appropriate intervals. Furthermore, the process generates a toggle signal for the UART receiver's operation. This toggle signal dictates which address (0-31 or 32-63) the BiPhase block transmits.

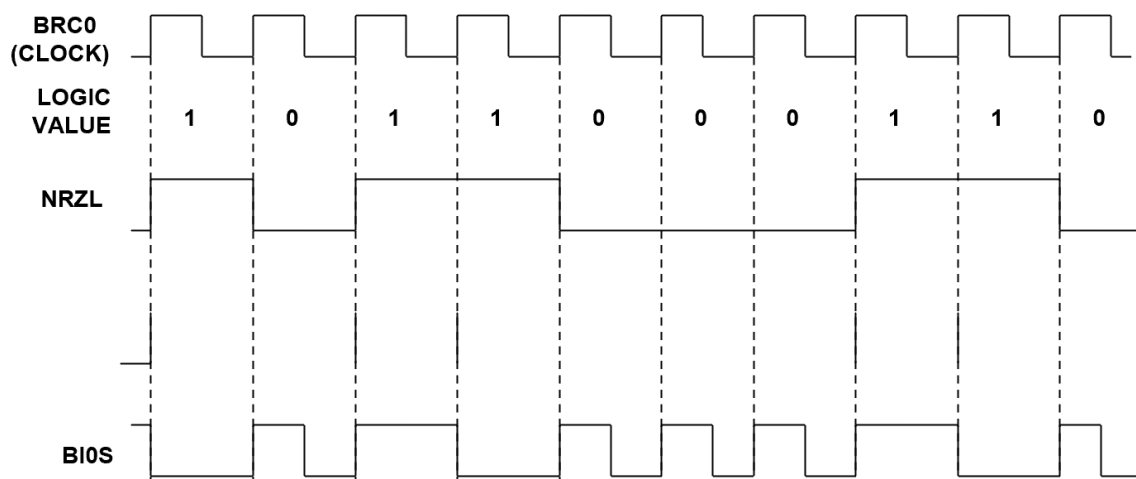
$$\text{Transmission Time} = 32 [\text{Bytes}] * 8 [\text{States}] * 328 * 10^{-6} [\text{Main Clock}] = 84ms \quad (3.9)$$

FSM flow chart is shown in Figure 3.17:



**Figure 3.17 BiPhase Transmitter Side FSM**

Overall, the Bi-Phase encoded output signal based on the rising and falling edges of the main clock and the value of the most significant bit of the data signal generated by the shift register operation. The process checks if there's a rising edge in the main clock signal or if there's a falling edge in the main clock signal while the data signal '0'. If either condition is met, the output signal inverted. An example of BiPhase encoder wave format is shown in Figure 3.18:

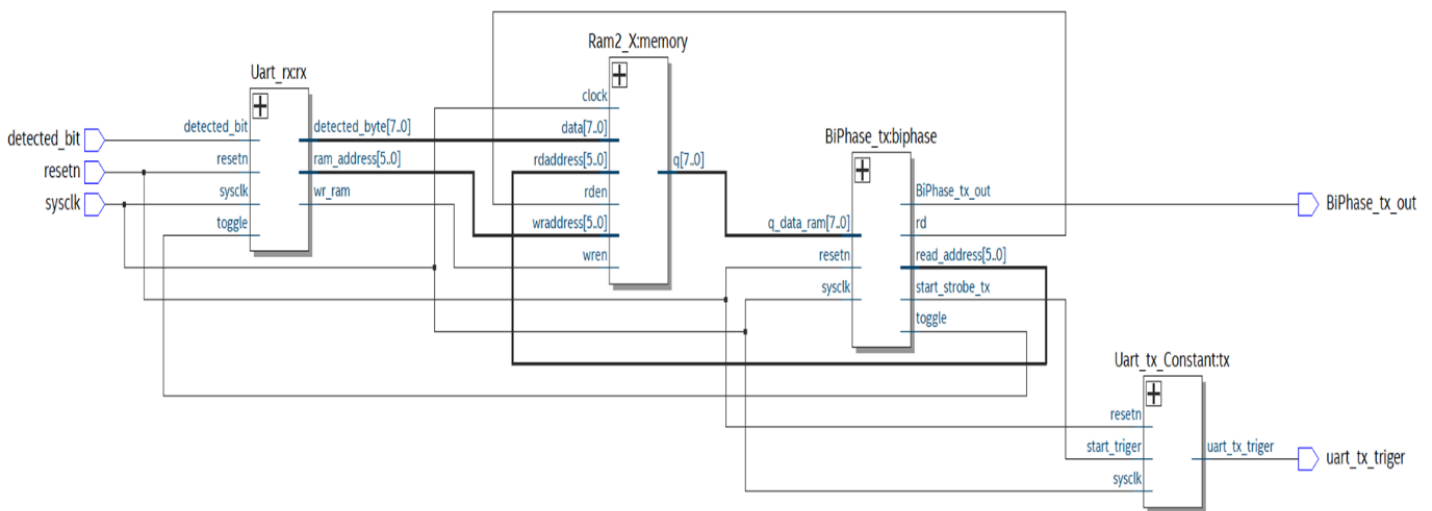


**Figure 3.18 BiPhase Encoder Wave Format**

### 3.2.9 Block 5: "Card A Design Python"

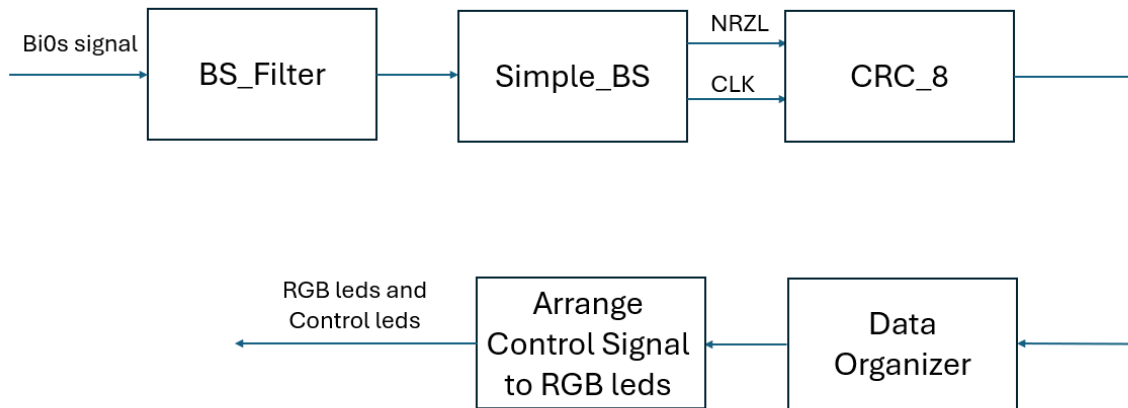
The "Card\_A\_Design\_Python" block serves as the central hub that integrates and connects various system blocks to form a hardware component focused on communication and data storage functionalities. This block role is to be linking together different modules such as UART communication, memory storage, 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\_trigger" and "BiPhase\_tx\_out", which are for coordinating the communication process and providing the encoded data for transmission.

The following diagram, shown in Figure 3.19, depicts the RTL connection diagram between the blocks of card A:



**Figure 3.19 Connection Diagram "Card\_A\_Design\_Python"**

### 3.3 Card B - short range transmission



**Figure 3.20 Card B short range block diagram**

In The Card B design incoming data, encoded in a bi-phase format, undergoes filtering via the "BS\_Filter" component. This state aims to refine the signal, potentially smoothing transitions or removing noise. The filtered output, 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, and conversion to a non-return-to-zero-level (NRZL) data format. Optionally, if CRC is enabled, the "CRC8BIT" component calculates an 8-bit checksum 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, formatting data for display on green LEDs and generating signals for RGB LED display. Finally, the RGB component takes charge, controlling the RGB LEDs based on the organized data and control signals. The following diagram, shown in Figure 3.20, depicts the card B block diagram.

### 3.3.1 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 3.21:



**Figure 3.21 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.

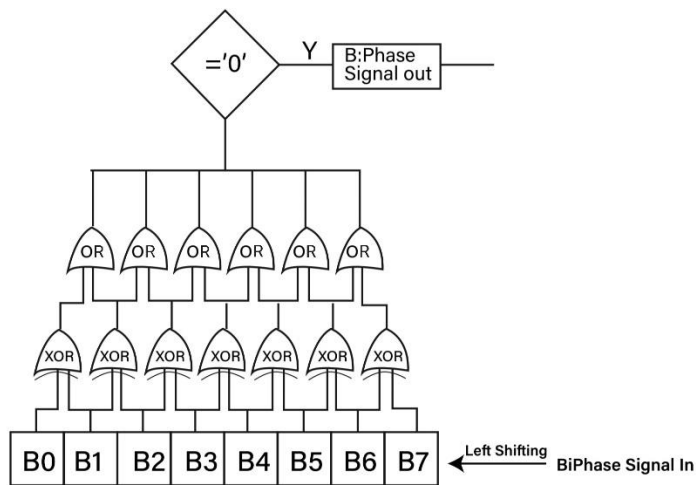
In our project the decoding BiPhase-encoded signal received by the SRX882 module, the purpose is to align the receiver's clock with the incoming signal and identify changes in voltage levels that represent the original data bits. Additionally, incorporating CRC into the data processing stage adds an 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. To extend the range of a wireless communication system, we can improve the signal-to-noise ratio. This be useful where long-distance communication is required.

### 3.3.2 Block 1: "BS\_Filter"

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

The architecture implements a debouncing filter using XOR operations on consecutive elements of the input signal record. 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 the filtered signal is updated with the current value of the biphas input signal.

A scheme of how-to filtering signals is shown in Figure 3.22:



**Figure 3.22 Filtering Signal**

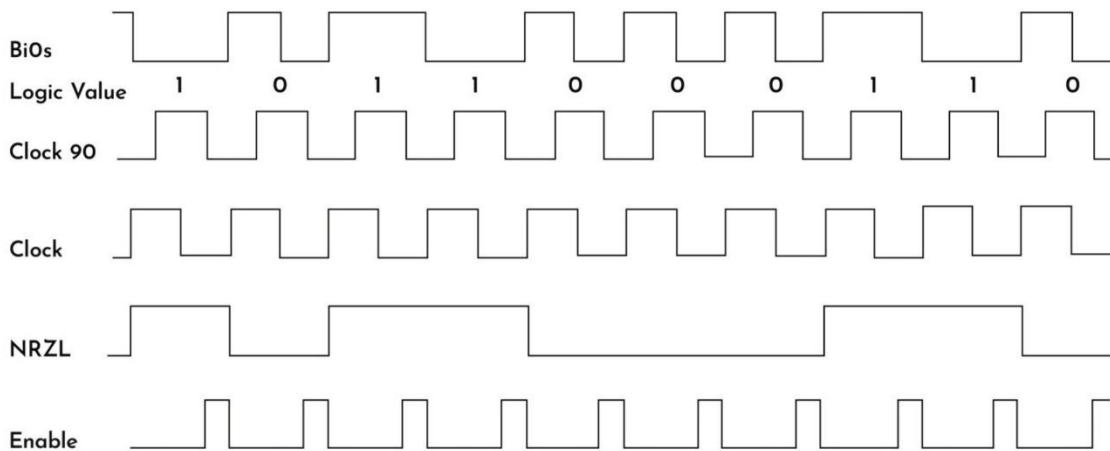


### 3.3.3 Block 2: "Simple\_BS"

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.

The architecture implements a biphas decoder to generate the original clock signal and a 90-degree shifted clock signal from the biphas signal. Using these clock signals, it sample the biphas signal on the rising edge of the clock signal shifted by 90 degrees and the original clock signal's logic 1, or on the falling edge of the clock signal shifted by 90 degrees and the original clock signal's logic 0. This process results in two samples per complete cycle, and decode the original information, an XNOR operation is performed on these two samples. Since the biphas coding operation follows XOR conditions, the XNOR operation successfully decodes the signal. An enable signal is utilized to synchronize the original clock signal with the biphas signal to accurate sampling of the biphas signal.

An example of BiPhase Decoder Wave Format is shown in Figure 3.23:



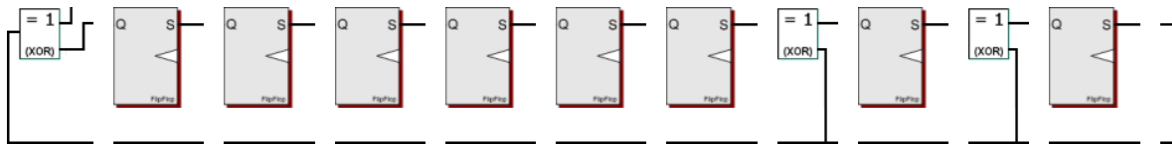
**Figure 3.23 BiPhase Decoder Wave Format**

### 3.3.4 Block 3: "CRC8BIT"

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

The architecture initiates a CRC check when receiving data from the NRZL signal. It first verifies that the header ("C0CAFEAB") is present in the initial 32 bits (4 bytes). The CRC check commences only after this header is successfully detected. The system uses autocorrelation, allowing data to pass as valid if at least 30 out of 32 header bits are correct. The algorithm operates by performing an XOR operation between the 32 bits of the pre-known header and the 32 incoming bits, identifying unmatched. This method enables the system to handle minor bit errors without requiring retransmission.

An example of CRC-8 RTL algorithm transitional diagram is shown in Figure 3.24:



**Figure 3.24 CRC-8 RTL Algorithm Transitional Diagram**

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. 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. Within the process, XOR operations are performed between the current CRC register value and the incoming data bit, updating each flip-flop accordingly. This process continues for each bit of the CRC register. Additionally, a counter 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.

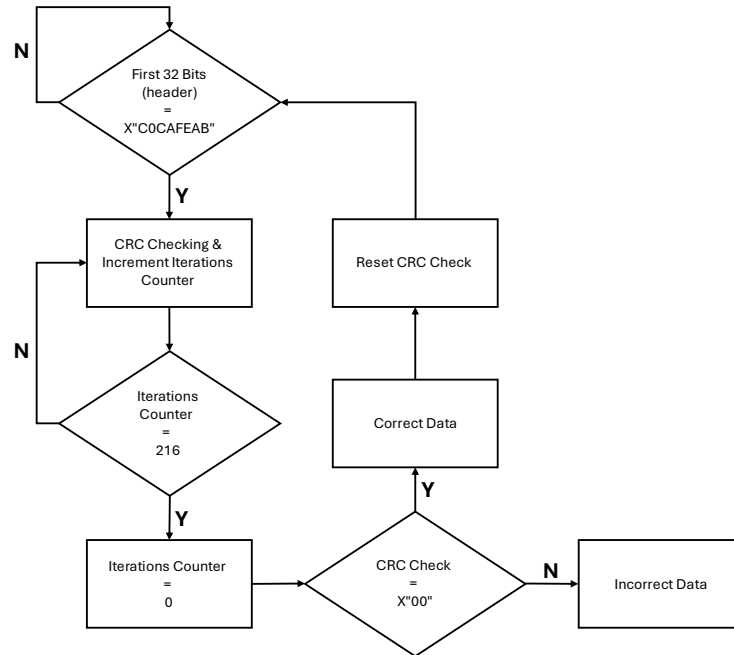
(3.10)

$$\begin{aligned} \text{counter calculation} &= (32(\text{Full data packet}) - 1(\text{the start byte}) - 4(\text{header})) * 8(\text{bits}) \\ &= 27 * 8 = 216 \text{ bits} \end{aligned}$$

$$P(x) = x^8 + x^2 + x^1 + x^0 \quad (3.11)$$

If the CRC check yields "X"00, the information is verified as correct. Otherwise, an error is detected, and the information will not proceed further in the process.

CRC-8 check operation flow chart is shown in Figure 3.25:



**Figure 3.25 CRC-8 Check Operation Flow Chart**

### 3.3.5 Block 4: "Data Organizer"

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

The architecture initiates a data organizer when receiving data from the NRZL signal. It first verifies that the header (X"CoCAFEAB") is present in the initial 32 bits (4 bytes). 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. The subsequent 12 bytes (96 bits) received from the NRZL signal are allocated for RGB LEDs, with each RGB LED using 3 bytes, as illustrated in Figure 3.26. An example of RGB frame composition is shown in Figure 3.26:

**Composition of 24bit data:**

G7	G6	G5	G4	G3	G2	G1	G0	R7	R6	R5	R4	R3	R2	R1	R0	B7	B6	B5	B4	B3	B2	B1	B0
----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----

**Figure 3.26 RGB Frame Composition**

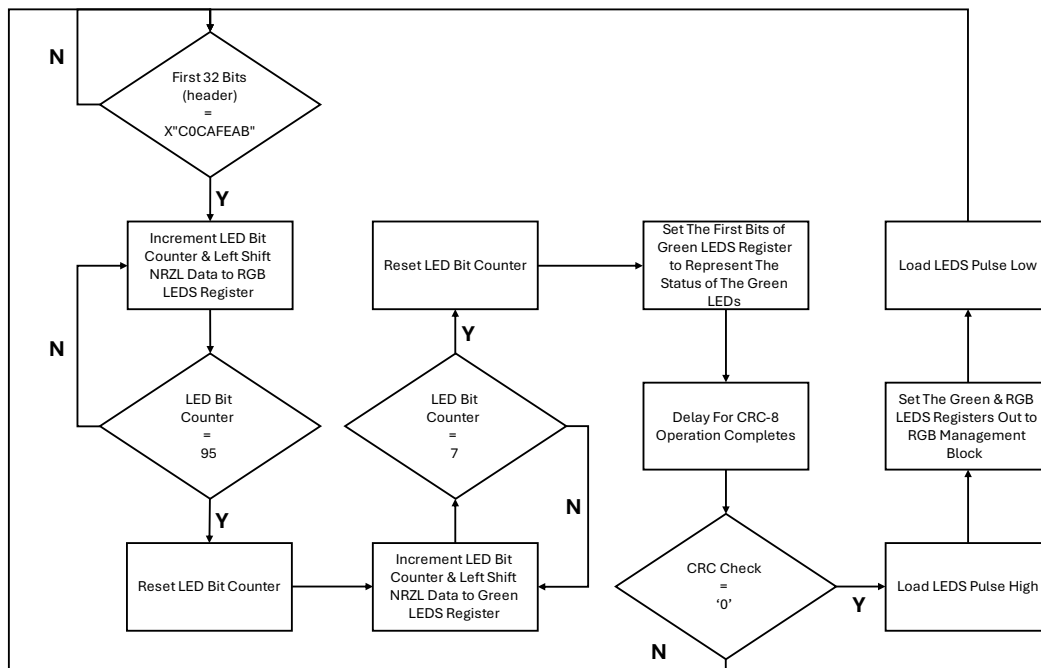
$$4 (RGB LEDs) * 24bit data = 96 bits \quad (3.12)$$

Following the RGB LEDs, the next byte is allocated for the green LEDs. Arranges and formats the received data to represent the status of the green LEDs when the first bit (LSB), second bit, and third bit are assigned to lights 1, 2, and 3, respectively. A delay of 110 main 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, so that the CRC operation completes before proceeding to the next stage of the process. The following calculation demonstrates the reasoning behind the delay according to the CRC Check process:

$$96(State0) + 8(S1) + 1(S2) + 1(S3) + 110(S4) = 216 (CRC8 - Operation) \quad (3.13)$$

The system evaluates the result of the CRC check. If the CRC result indicates that the data is error-free, the system proceeds to load the LEDS and manages the timing for transferring the data to the RGB and green LEDs on the FPGA card.

Data organizer flow chart is shown in Figure 3.27:



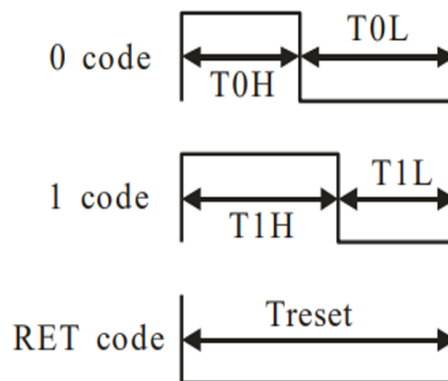
**Figure 3.27 Data Organizer Operation**

### 3.3.6 Block 5: "RGB"

This block is designed to control RGB LEDs, 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.

The architecture initiates a LEDs setup. Initially, the status of green LEDs is assigned to LED\_1, LED\_2 and LED\_3 respectively to the first (LSB to MSB) bits. Then, a single bit is output from the RGB LEDs Register at a time. Each bit is evaluated to determine if it is a logical 0 or 1. Based on this evaluation, the appropriate waveform is generated, as illustrated in Figure 3.28 of the datasheets, to produce the OB\_LED\_RGB\_DIN signal as shown in Figure 3.30 below. The waveform timings are selected according to the specifications in Figure 3.29 of the datasheets. For a bit '1', the signal maintains a logical 1 state for 740ns, followed by a logical 0 state for 360ns. Conversely, a bit '0' maintains a logical 0 state for 800ns, followed by a logical 1 state for 300ns. This timing is consistent with the specifications outlined in the datasheet.

The logic 1 or 0 of each bit is determined as shown in Fig 3.28. It is determined by the length of the voltage level in a period. Detailed specifications can refer to Fig 3.29.

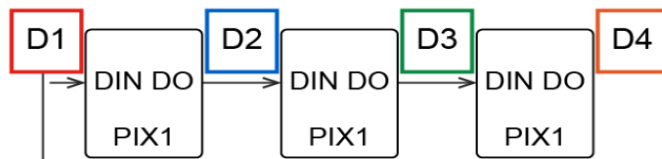


**Figure 3.28 Code Timing**

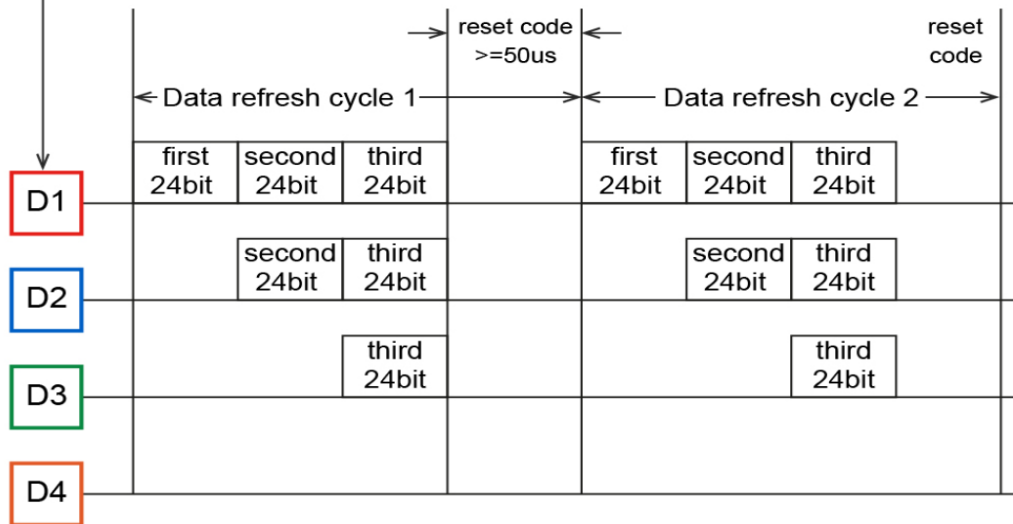
Code Status	Description	Timing
T0H	0 code, high voltage time	220ns~380ns
T1H	1 code, high voltage time	580ns~1μs
T0L	0 code, low voltage time	580ns~1μs
T1L	1 code, low voltage time	220ns~420ns
RES	Frame unit, low voltage time	>280μs

**Figure 3.29 Sequence Chart**

## Cascade method:



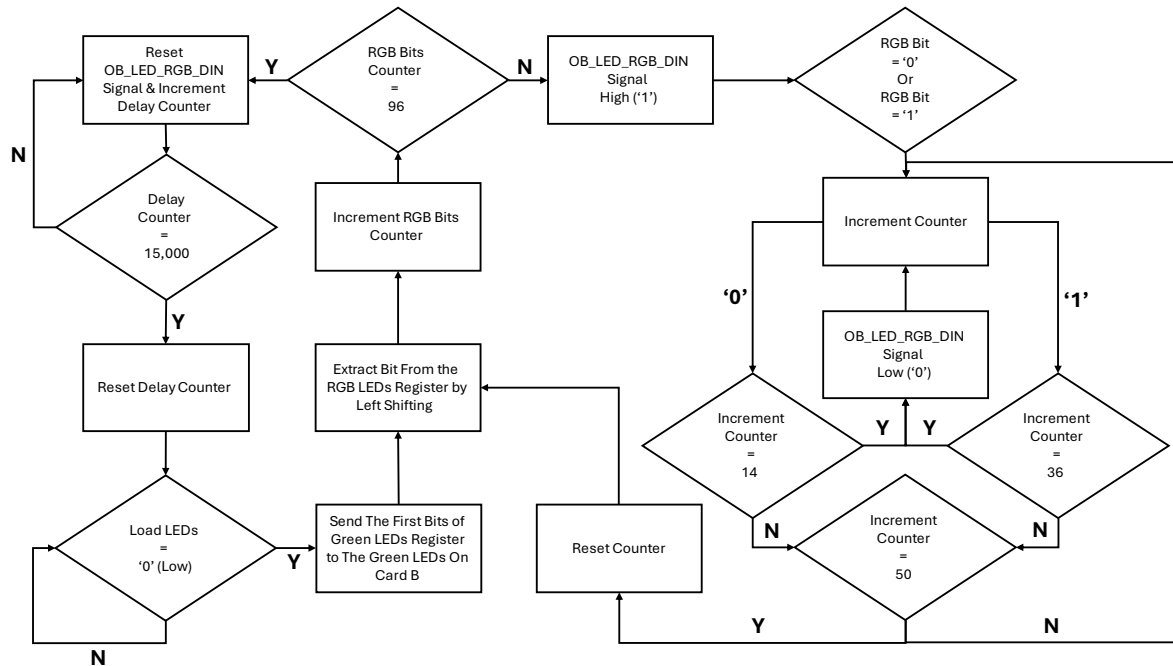
## Data transmission method:



**Figure 3.30 Data Transmission Method**

The data transfer protocol of the WS281 use single NRZ communication mode. As shown in Fig 3.27, after the pixel power-on reset, the DI port receive data from controller ("OB\_LED\_RGB\_DIN" output), the first pixel collect initial 24-bit data then sent to the internal data latch, the other data which reshaping by the internal signal reshaping amplification circuit sent to the next cascade pixel through the DO port. After transmission for each pixel, the signal to reduce 24bit. Pixel adopt auto reshaping transmit technology, making the pixel cascade number is not limited the signal transmission, only depends on the speed of signal transmission.

LEDs Management flow chart is shown in Figure 3.31:

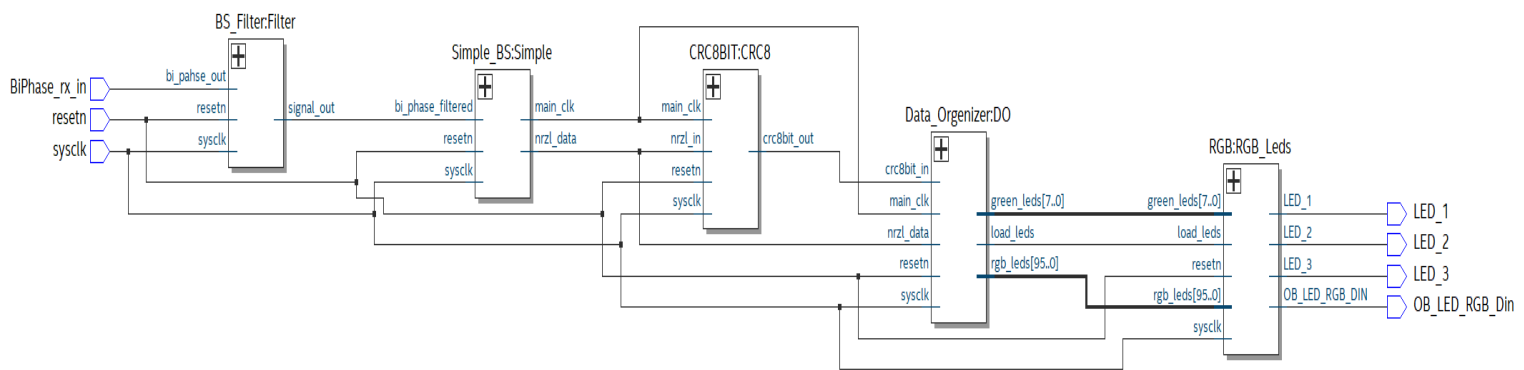


**Figure 3.31 LEDs Management**



### 3.3.7 Block 6: "Card\_B\_Design"

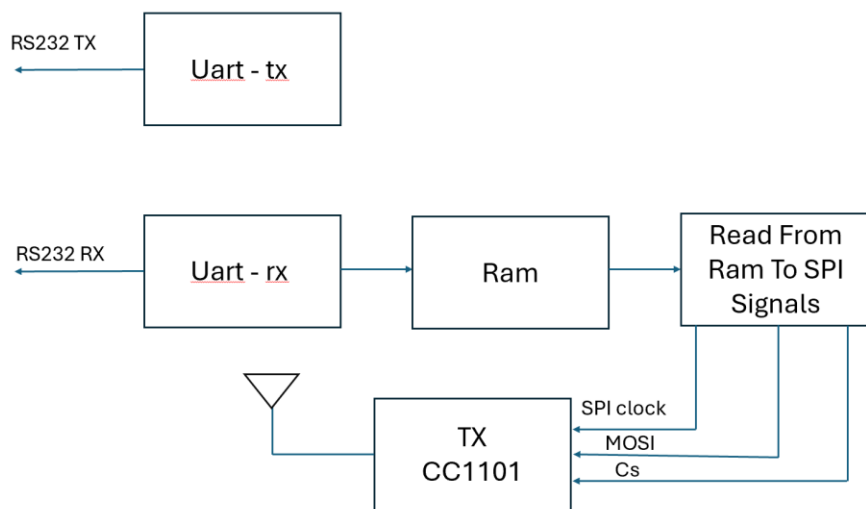
The "Card\_B\_Design" block embodies the entire process of receiving, decoding, organizing, and displaying data within the system architecture. It begins by interfacing with the SRX882 module to decode the data stream wirelessly transmitted by the STX882 transmitter. Once the data is decoded, "Card\_B\_Design" systematically arranges it before displaying it on an array of LEDs and an RGB display. 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 3.32, depicts the RTL connection diagram between the blocks of card B:



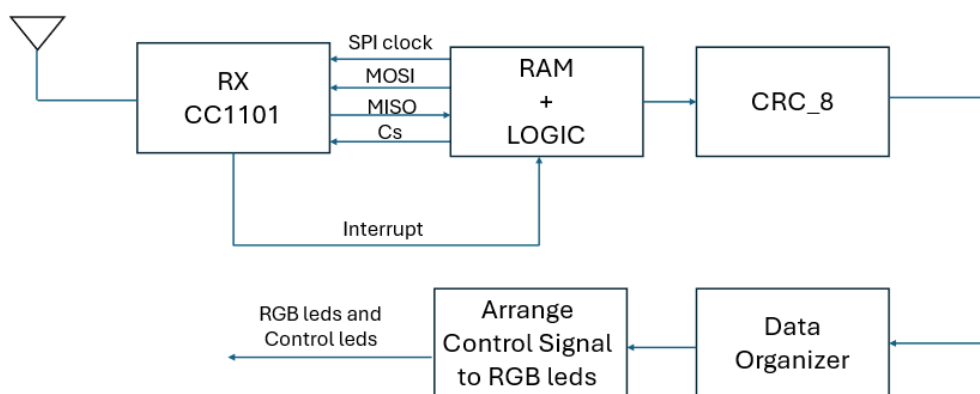
**Figure 3.32 Connection Diagram "Card\_B\_Design"**

## 3.4 SPI - long range transmission

We employ the Serial Peripheral Interface (SPI) protocol as a second option for communication between the FPGA and the CC1101 transceiver module. SPI is a widely-used synchronous data transfer standard that enables high-speed communication between microcontrollers and peripheral devices. It operates on a master-slave architecture, where the master device controls the data flow, and the slave device responds accordingly. The protocol uses four main signals: Master Out Slave In (MOSI), Master In Slave Out (MISO), Serial Clock (SCLK), and Chip Select (CS). The following diagrams, shown in Figure 3.33 and Figure 3.34, illustrating the structure of Cards A and B, most of the system components remain unchanged, with the exception of the front-end transmission parts, which have been replaced by CC1101 blocks:

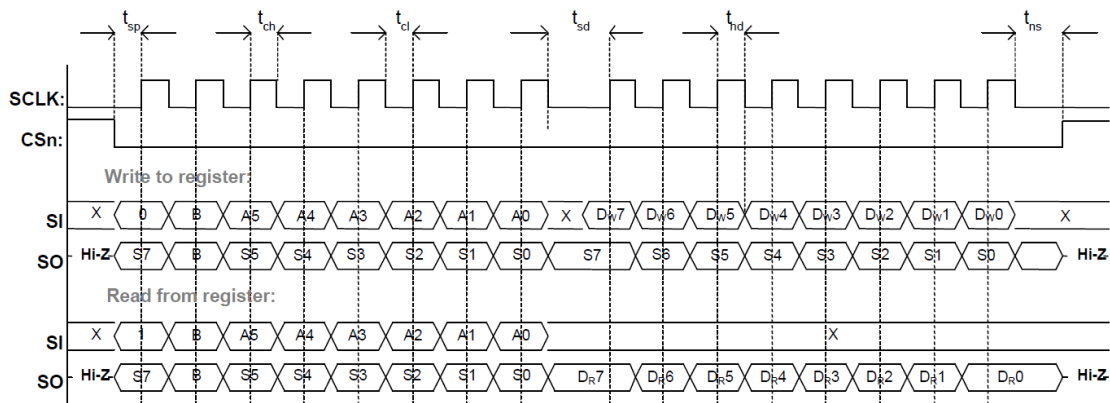


**Figure 3.33 Card A long range block diagram**



**Figure 3.34 Card B long range block diagram**

In our project, we employ the Serial Peripheral Interface (SPI) protocol as a second option for communication between the FPGA and the CC1101 transceiver module. SPI is a widely-used synchronous data transfer standard that enables high-speed communication between microcontrollers and peripheral devices. It operates on a master-slave architecture, where the master device controls the data flow, and the slave device responds accordingly. Configuration Registers Write and Read Operations shown in Figure 3.35:



**Figure 3.35 Configuration Registers Write and Read Operations**

The CC1101 transceiver module, which we use for long-distance communication, utilizes the SPI protocol to interface with the FPGA. The FPGA, acting as the master, sends and receives data to and from the CC1101 module. The data transfer process begins with the FPGA generating the clock signal (SCLK) to synchronize communication. The MOSI line carries data from the FPGA to the CC1101, while the MISO line transmits data back from the CC1101 to the FPGA. The CS line selects the CC1101 transceiver, enabling it to communicate with the FPGA.

In chapters 3.4, we will delve into the processes of writing information to the CC1101 transmitter and reading information from the CC1101 receiver. However, we have not yet covered the communication between the transmitter and receiver.

## 3.4.1 spi\_cc1101\_write

Writing data to the CC1101 transceiver using SPI communication involves a structured sequence of steps managed by a state machine. The chip select (CS) line is then pulled low to activate the CC1101's SPI interface, signaling that communication is about to commence.

Once SPI is activated, the state machine begins sending the data bit by bit through the MOSI line. A clock signal, generated by the state machine, synchronizes the transmission of each bit. As each bit is sent, the data register shifts, and all bits are transmitted in the correct order. Throughout this process, the system continuously monitors and updates the internal state to manage timing and ensure data integrity. After the complete data byte is transmitted, the cc1101 receiving the write command, when the MSB of this byte indicates whether the operation is a write or read. The bit immediately to the right of the MSB signifies whether the operation is a single-byte transaction or a burst mode transaction. The remaining six bits specify the address of the register where the data will be written.

Once this command byte is processed, the system understands which register to write to and the type of operation. The next byte received is the actual data to be written into the specified register. This pattern continues, with every pair of bytes consisting of a command byte. the CS line is then pulled high, ending the SPI communication. SPI writing module FSM flow chart is shown in Figure 3.36:

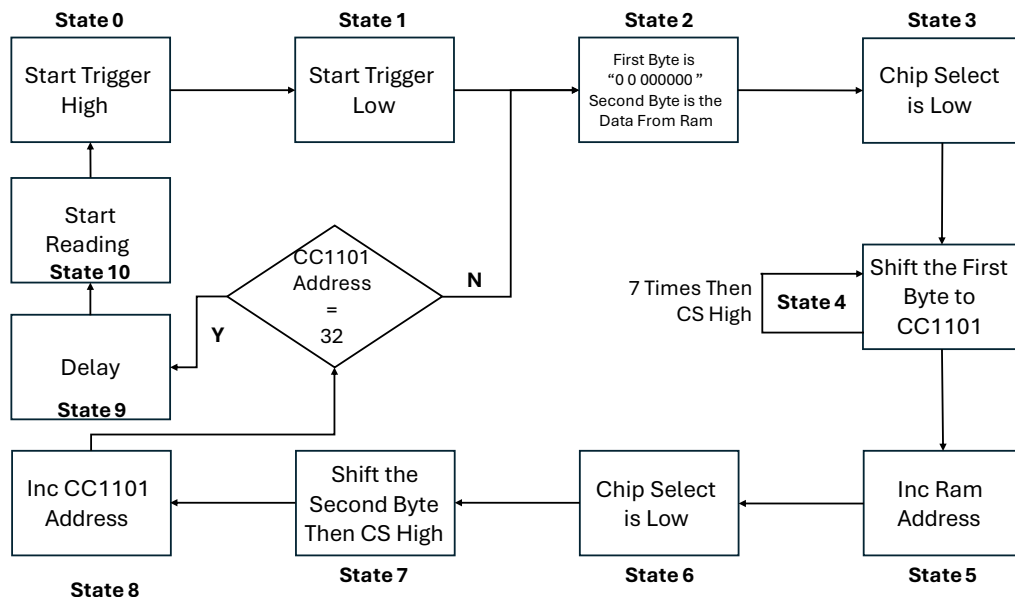
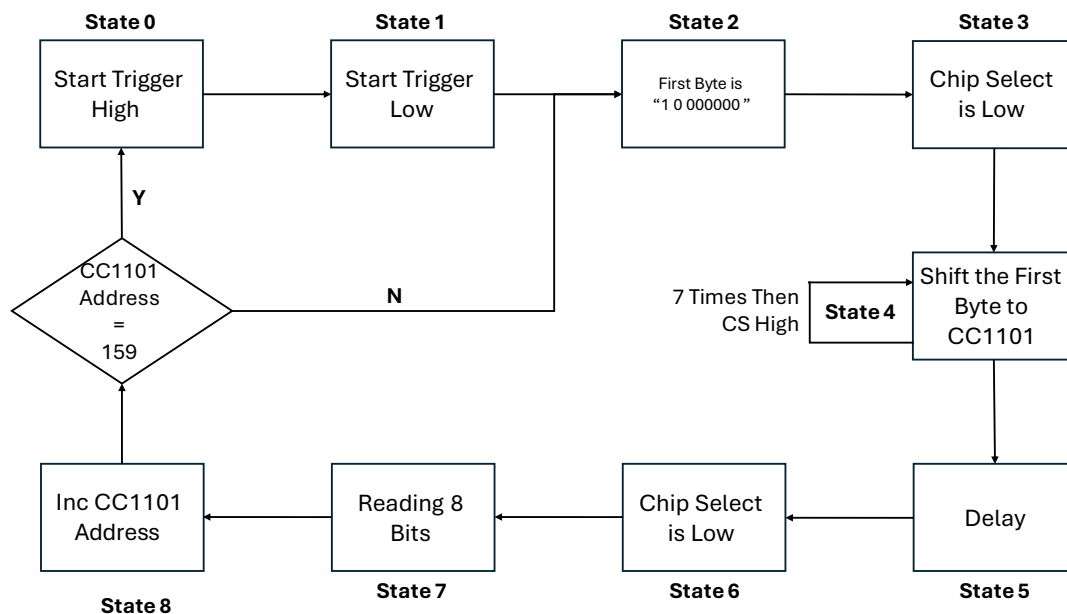


Figure 3.36 SPI Write Module FSM

## 3.4.2 spi\_cc1101\_read

Reading data from the CC1101 transceiver using SPI communication involves a structured sequence of steps managed by a state machine. The chip select (CS) line is then pulled low to activate the CC1101's SPI interface, signaling that communication is about to commence.

The reading process is similar to the writing process, but with a key difference in the command byte. When initiating a read operation, the MSB (most significant bit) of the first byte is set to 1, which signals that the command is a read request. After this command byte is sent, the CC1101 responds by providing the data from the specified register. The data is read one byte at a time, corresponding to the address specified in the command. This process continues, with each command byte leading to a subsequent data byte being read from the CC1101 register. SPI reading module FSM flow chart is shown in Figure 3.37:



**Figure 3.37 SPI Write Module FSM**

#### 4. 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.

## 4.1 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:

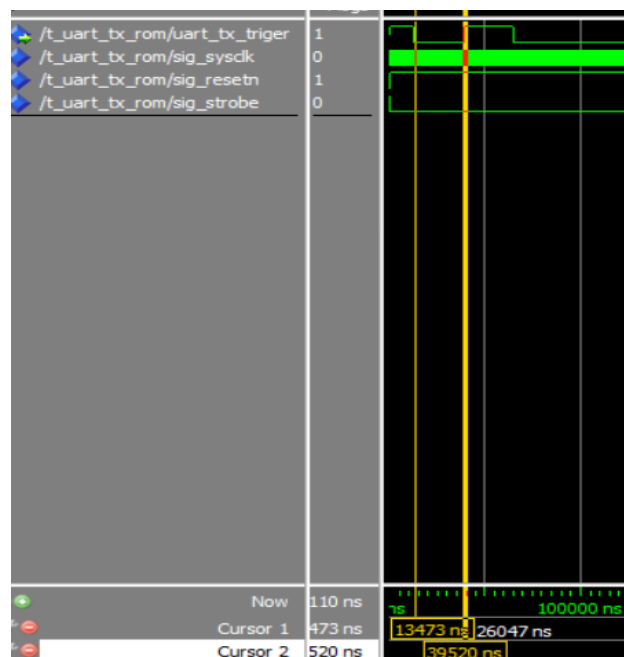
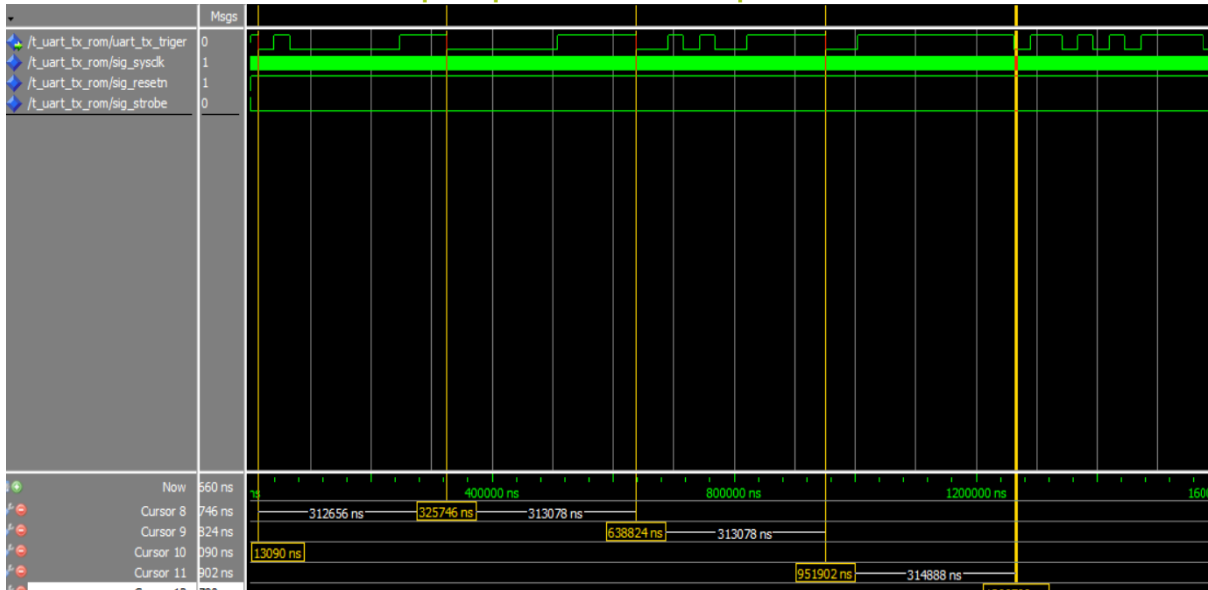


Figure 4.1 Uart\_tx simulation start

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. The simulation of "Uart\_tx" data transfer is shown in Figure 4.2:

## המחלקה להנדסת חשמל ואלקטרוניקה



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

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

```
depth = 32;
width = 8;
address_radix = HEX;
data_radix = hex;
content
begin

0:01;
1:C0;
2:CA;
3:FE;
4:AB;

5:00;
6:22;
```

**Figure 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.



## 4.2 Test Bench 2: "Uart\_rx"

To test the "uart\_rx" block, we use the "Test\_uart\_rx" code, which connects "uart\_tx\_rom", "uart\_rx", and "Ram2\_X". This setup checks data integrity during transfer from ROM to "uart\_tx\_rom", then to "uart\_rx", and finally to "Ram2\_X", validating the communication system's reliability. The simulation of "Uart\_rx" data transfer is shown in Figure 4.4:



Figure 4.4 "Uart\_rx" Simulation data transfer

From the figure provided, each byte takes 307,200 nanoseconds to transfer, excluding the initial "01" byte's stabilization period. The "q\_ram" output represents the RAM output during sequential data transfer, while "detected\_byte" is the output of "uart\_rx", containing all received data from the ROM. This simulation demonstrates successful serial communication at 38400 Hz using VHDL, connecting the computer and FPGA card.

## 4.3 Test Bench 3: "Card\_A\_Design"

To test the "BiPhase\_tx" block and "Card\_A\_Design\_Python", we use the "Card\_A\_Design" code, which connects "Uart\_tx\_Rom", "Uart\_rx", "Ram2\_X", and "BiPhase\_tx". This setup ensures data integrity during transfer from ROM to "uart\_tx", then to RAM, "uart\_rx", and finally "BiPhase\_tx". The simulation of "Card\_A\_Design" data transfer to "BiPhase\_tx\_out" is shown in Figure 4.5:

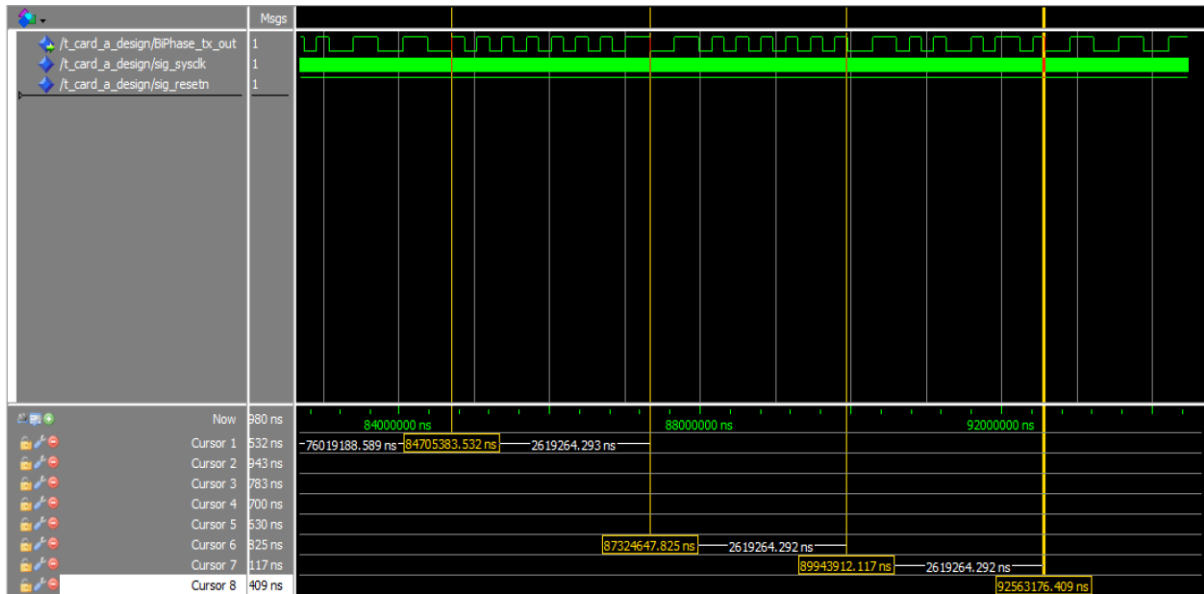
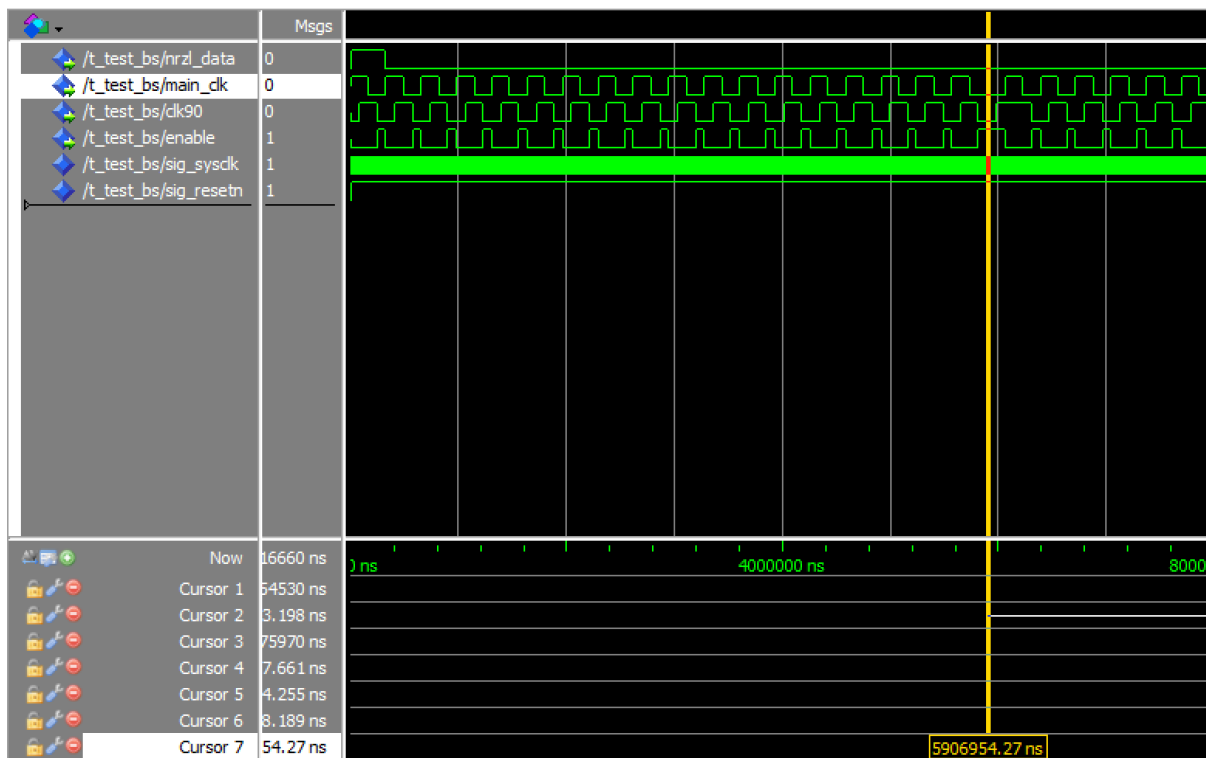


Figure 4.5 "Card\_A\_Design" Simulation data transfer to "BiPhase\_tx\_out"

From the figure provided, each bit takes 327,680 nanoseconds to travel. The output "BiPhase\_tx\_out" shows the "BiPhase\_tx" block's output, demonstrating the expected BiPhase encoding with alternating high and low states for logical '0's and '1's. The pattern matches the BiPhase coding scheme, confirming accurate signal encoding and decoding. The simulation output, with "01" followed by "CO" and "CA" as specified in the MIF file, confirms reliable data transmission within the system.

#### 4.4 Test Bench 4: "Test\_BS"

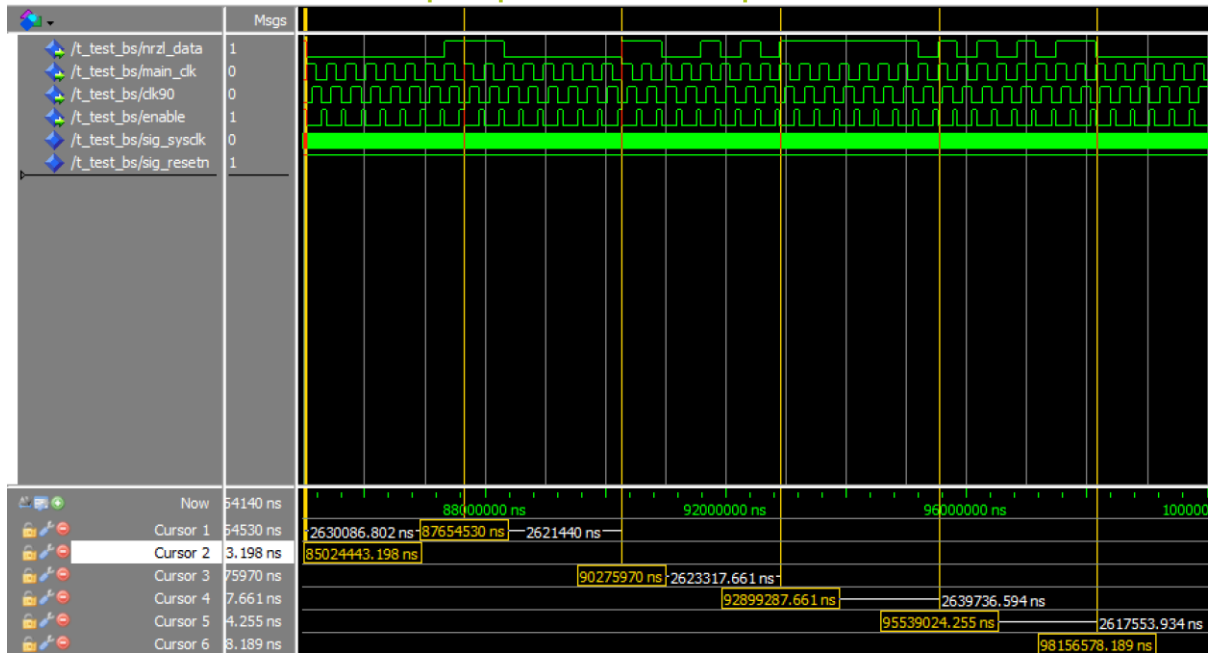
To test the "BS\_Filter" and "Simple\_BS" blocks, we use the "Test\_BS" code, which connects "Card\_A\_Design", "BS\_Filter", and "Simple\_BS". This setup checks data integrity during transfer from "Card\_A\_Design" to "BS\_Filter" for filtering and then to "Simple\_BS" for decoding. Synchronization simulation between "nrzl\_data" and "main\_clk" is shown in Figure 4.6:



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

In the simulation, The cursor marks the synchronization point where the first logical 1 is used for alignment. This ensures the data reception aligns with the clock signal. Simulation of "BS\_Filter" and "Simple\_BS" data transfer as "nrzl\_data" port is shown in Figure 4.7:

## המחלקה להנדסת חשמל ואלקטרוניקה

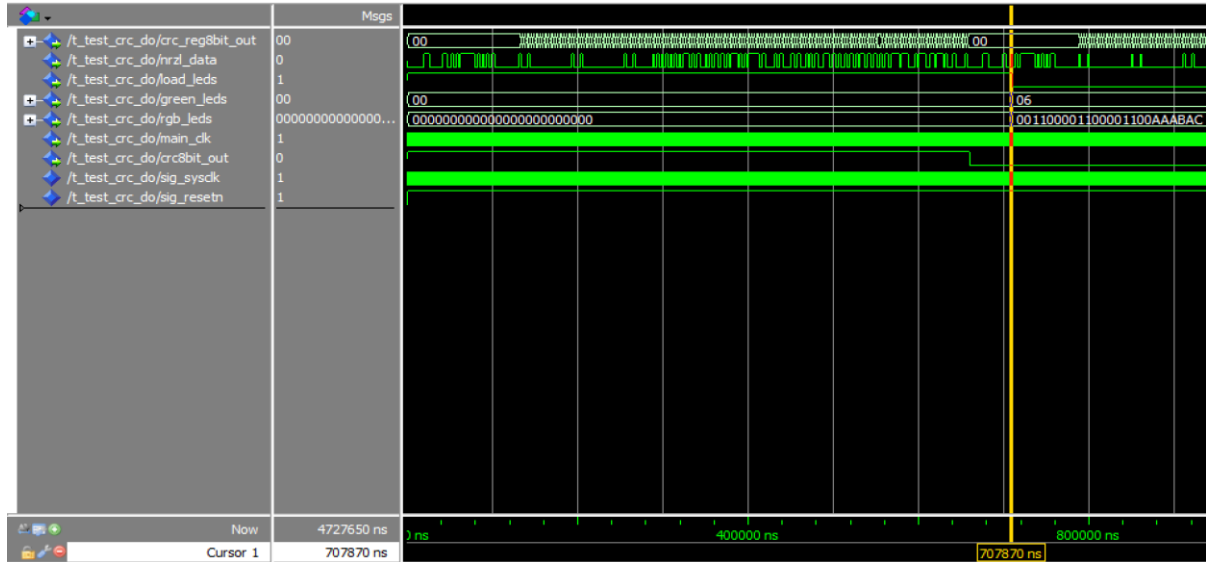


**Figure 4.7 "BS\_Filter" & "Simple\_BS" Simulation data transfer to "nrzl\_data"**

From the figure provided, Each bit takes 327,680 nanoseconds to travel. The outputs "nrzl\_data" and "main\_clk" from the "Simple\_BS" block show the "BS\_Filter" effectively isolates the "BiPhase\_tx\_out" signal from Card A, removing transmission noise. The decoded "nrzl\_data" shows the sequence "01", "CO", "CA", "FE", and "AB". The "main\_clk" signal synchronizes accurately with the information signal, validating the process. The shifted "clk90" signal, with a 90-degree phase shift, aligns with the main clock's midpoints, ensuring precise sampling of the "BiPhase\_tx\_out" signal. The simulation confirms the system design's accuracy and reliability.

## 4.5 Test Bench 5: "Test\_CRC\_DO"

To test the "CRC8BIT" block and "Data\_Organizer", we use the "Test\_CRC\_DO" code, connecting "ROM\_CRC", "CRC8BIT", and "Data\_Organizer". This setup checks data integrity from "ROM\_CRC" to "CRC8BIT" for CRC checking, and finally to "Data\_Organizer". Integrity check simulation of CRC Check and Data Organize ports are shown in Figure 4.8:



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

The output signals "crc\_reg8bit\_out" and "crc8bit\_out" come from "CRC8BIT", while "load\_leds", "green\_leds", and "rgb\_leds" come from "Data\_Organizer". The CRC check ("crc\_reg8bit\_out") shows a normal result ("00"), matching the manual CRC calculation stored in the last byte of memory. The "crc8bit\_out" signal turns logical 0, indicating normal information for the "Data\_Organizer" block. Upon this normal CRC result, the "load\_leds" pulse transfers correct data to "green\_leds" and "rgb\_leds". The simulation confirms orderly transfer, reflecting the pre-planned memory in the LED signals, validating data integrity and successful simulation.

## 4.6 Test Bench 6: "Card\_B\_Design"

To test the "Card\_B\_Design" block, we use the "Card\_B\_Design" code, connecting six blocks: "BiPhase\_Generator", "BS\_Filter", "Simple\_BS", "CRC8BIT", "Data\_Organizer", and "RGB". This setup checks data integrity from "BiPhase\_Generator" to "RGB" for LED manipulation. Integrity check simulation of the system output is shown in Figure 4.9:

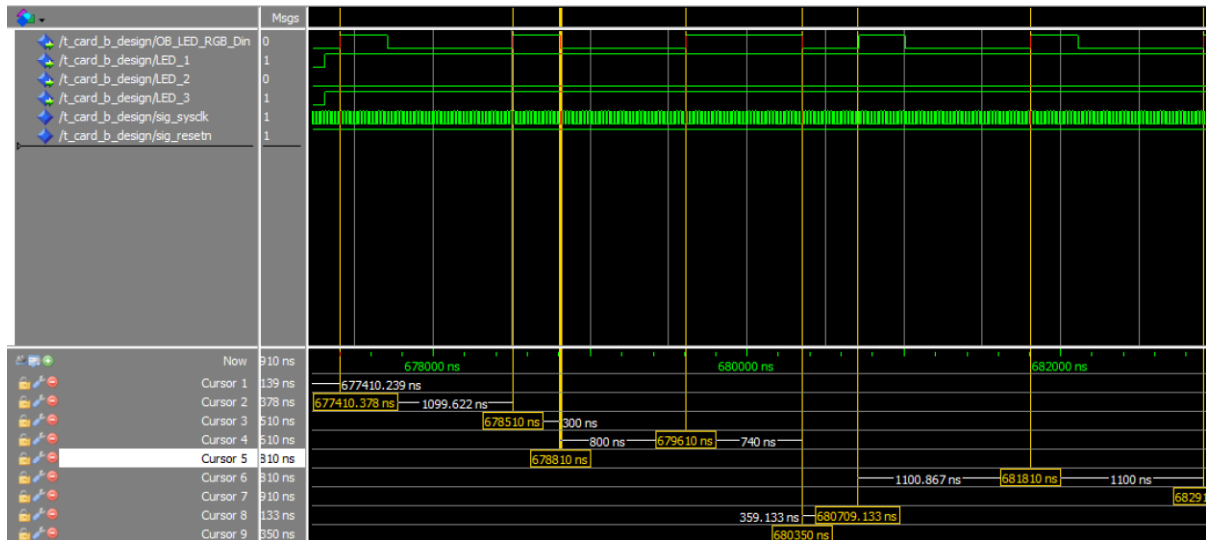
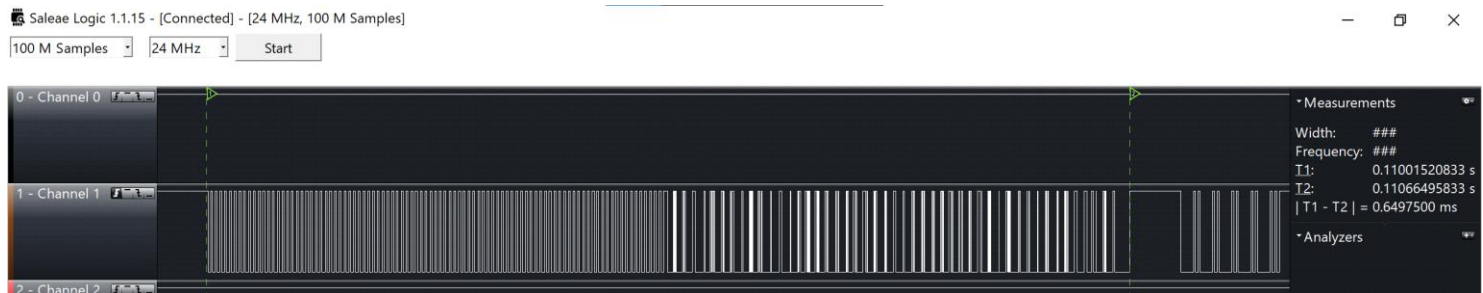


Figure 4.9 "Card\_B\_Design" Simulation, The System Output

The simulation shows each bit transmitted to the RGB LEDs via "OB\_LED\_RGB\_DIN" follows a precise timing pattern of 1100ns per bit. A bit '1' has a logical 1 state for 740ns followed by a logical 0 for 360ns, while a bit '0' has a logical 0 for 800ns followed by a logical 1 for 300ns, matching the datasheet specifications. The green LEDs accurately reflect the memory data, with two LEDs on and one off. This confirms the information is transmitted correctly, validating the data transmission process and the system's integrity.

#### 4.7 Logic Analyzer: "spi\_cc1101\_read/write" check

To test the reading and writing accuracy of the CC1101 component, we wrote predetermined values and then read them back to verify correctness. Each operation, writing and reading, takes 0.325ms. Writing and reading process is shown in figure 4.10:



**Figure 4.10 Writing and Reading Process Shown in Logic Analyzer**

The figure 4.10 shows the complete process between the two markers: writing is done in the first half and reading in the second half, reading from CC1101 is shown in Figure 4.11.



**Figure 4.11 Reading from CC1101**

The results confirm that we read the bytes from the specified register and received the exact values we wrote, verifying the correctness of the reading and writing process.

## 5. Conclusion

Wireless communication systems based on FPGA cards benefit industries like defense and automotive due to their reliability and flexibility. This project aims to create such a system, involving the design of system architecture and communication protocols. The setup allows wireless communication between two FPGA cards, Card A and Card B, with a computer application controlling data transmission. Card A sends data wirelessly to Card B, which adjusts LED intensity accordingly. The system offers short-range communication using STX882 and SRX882 with BiPhase coding and long-range using CC1101 with SPI coding. Noise reduction filters, Debouncer, and CRC-8 error detection enhance reliability, and MAX 10 cards increase speed. Testing is comprehensive, with real-time data transmission showing close alignment with theoretical calculations. However, low-frequency signals require higher simulation frequencies or more powerful computers. Transmission speed decreases with distance; short-range tests showed delays beyond 60 meters. Future iterations will include an encryption unit to ensure data confidentiality and integrity, crucial for industries with stringent security requirements.



## 6. References

- [1] R. Douglass, K. Gremban, A. Swami, and S. Gerali, Eds., “IoT for Defense and National Security,” in *IoT for Defense and National Security*, 1st ed., Wiley, 2022. doi: 10.1002/9781119892199.fmatter.
- [2] R. Ferdian, R. Aisuwarya, and T. Erlina, “Edge Computing for Internet of Things Based on FPGA,” in *2020 International Conference on Information Technology Systems and Innovation (ICITSI)*, Bandung - Padang, Indonesia: IEEE, Oct. 2020, pp. 20–23. doi: 10.1109/ICITSI50517.2020.9264937.
- [3] T. Akilan, S. Chaudhary, P. Kumari, and U. Pandey, “Surveillance Robot in Hazardous Place Using IoT Technology,” in *2020 2nd International Conference on Advances in Computing, Communication Control and Networking (ICACCCN)*, Greater Noida, India: IEEE, Dec. 2020, pp. 775–780. doi: 10.1109/ICACCCN51052.2020.9362813.
- [4] S. K. Das, *Mobile handset design*. Singapore ; Hoboken, NJ: John Wiley & Sons (Asia), 2010.
- [5] “MIXED SIGNAL VLSI WIRELESS DESIGN”.
- [6] “Trust-Based Communication Systems for Internet of Things Applications - 2022 - Agrawal (1).pdf.”
- [7] K. Iniewski, C. McCrosky, and D. Minoli, *Network Infrastructure and Architecture: Designing High-Availability Networks*, 1st ed. Wiley, 2008. doi: 10.1002/9780470253526.
- [8] “VLSI\_Implementation\_of\_8-PSK\_8-QAM\_16-QAM\_Transmitter\_Circuit\_in\_CMOS\_Technology”.
- [9] 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.
- [10] 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.
- [11] 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.
- [12] 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.
- [13] 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.
- [14] 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.
- [15] D. Sundararajan, “Introduction,” in *Control Systems*, Cham: Springer International Publishing, 2022, pp. 1–14. doi: 10.1007/978-3-030-98445-8\_1.

- [16] J. A. Dell, *Digital Interface Design and Application*, 1st ed. Wiley, 2015. doi: 10.1002/9781119107507.
- [17] 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.
- [18] 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.
- [19] A. Loan, "Line Coding," in *Handbook of Computer Networks*, 1st ed., H. Bidgoli, Ed., Wiley, 2007, pp. 522–537. doi: 10.1002/9781118256053.ch34.
- [20] 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.
- [21] 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.

## 7. Appendices

### 7.1 Python Interface Code:

```
from os import stat ##### 01.08.2024 last time changes to fram 32 bytes
from tkinter import *
import os
import serial
import customtkinter
import time

#ser = serial.Serial("COM3", 38400)

ser = serial.Serial(
    "COM5",
    baudrate=38400,
    parity=serial.PARITY_NONE,
    stopbits=serial.STOPBITS_TWO,
    bytesize=serial.EIGHTBITS
)

array = [0x01, 0xC0, 0xCA, 0xFE, 0xAB]

customtkinter.set_appearance_mode("System")
root = Tk()
root.geometry('850x700')

horizontal1 = ""
horizontal2 = ""
horizontal3 = ""
horizontal4 = ""
horizontal5 = ""
horizontal6 = ""
horizontal7 = ""
horizontal8 = ""
horizontal9 = ""
horizontal10 = ""
horizontal11 = ""
horizontal12 = ""
flag = b'\xca'

buttons = 0
```

```
def updateArray():
    global array, flag

    array = [0x01, 0xC0, 0xCA, 0xFE, 0xAB]

    array.extend([int(horizontal1.get()), int(horizontal2.get()), int(horizontal3.get()),
int(horizontal4.get()),
                int(horizontal5.get()), int(horizontal6.get()), int(horizontal7.get()), int(horizontal8.get()),
                int(horizontal9.get()), int(horizontal10.get()), int(horizontal11.get()), int(horizontal12.get()),
                buttons])

    for i in range(1, 15): # for loop that appends 0x00 from slot 19 (included) to 32

        array.append(0x00)

    # Example usage with an array of integers representing hexadecimal values:
    hex_data = array[5: 31]
    crc_value = calculate_crc8(hex_data)
    array[-1] = crc_value
    print(f"CRC8 checksum: 0x{crc_value:02X}")

    print(array)
    root.after(30)

    ser.write(array)

    s = ser.read(1)

    flag = s

def txrx():
    ser.write(array)
    s = ser.read(1)
    flag = s

def createLabels():
```

```
#####
```

```
##Led #1
```

```
global horizontal1, horizontal2, horizontal3, horizontal4
```

```
global horizontal5, horizontal6, horizontal7, horizontal8
```

```
global horizontal9, horizontal10, horizontal11, horizontal12
```

```
horizontal1 = Scale(root, from_=0, to=255, bg="green", fg="white", font=('Helvetica 18 bold',15))
```

```
horizontal1.place(x=100, y=50, height=200)
```

```
horizontal2 = Scale(root, from_=0, to=255, bg="red", fg="white", font=('Helvetica 18 bold', 15))
```

```
horizontal2.place(x=200, y=50, height=200)
```

```
horizontal3 = Scale(root, from_=0, to=255, bg="blue", fg="white", font=('Helvetica 18 bold', 15))
```

```
horizontal3.place(x=300, y=50, height=200)
```

```
#####
```

```
##Led #2
```

```
horizontal4 = Scale(root, from_=0, to=255, bg="green", fg="white", font=('Helvetica 18 bold',15))
```

```
horizontal4.place(x=500, y=50, height=200)
```

```
horizontal5 = Scale(root, from_=0, to=255, bg="red", fg="white", font=('Helvetica 18 bold', 15))
```

```
horizontal5.place(x=600, y=50, height=200)
```

```
horizontal6 = Scale(root, from_=0, to=255, bg="blue", fg="white", font=('Helvetica 18 bold', 15))
```

```
horizontal6.place(x=700, y=50, height=200)
```

```
#####
```

```
## Led #3
```

```
horizontal7 = Scale(root, from_=0, to=255, bg="green", fg="white", font=('Helvetica 18 bold',15))
```

```
horizontal7.place(x=100, y=350, height=200)
```

```
horizontal8 = Scale(root, from_=0, to=255, bg="red", fg="white", font=('Helvetica 18 bold', 15))
```

```
horizontal8.place(x=200, y=350, height=200)
```

```
horizontal9 = Scale(root, from_=0, to=255, bg="blue", fg="white", font=('Helvetica 18 bold', 15))
```

```
horizontal9.place(x=300, y=350, height=200)
```

```
#####
```

```
##Led #4
```

```
horizontal10 = Scale(root, from_=0, to=255,bg="green",fg="white",font=('Helvetica 18 bold',15))
horizontal10.place(x=500, y=350, height=200)
```

```
horizontal11 = Scale(root, from_=0, to=255, bg="red", fg="white",font=('Helvetica 18 bold', 15))
horizontal11.place(x=600, y=350, height=200)
```

```
horizontal12 = Scale(root, from_=0, to=255,bg="blue", fg="white",font=('Helvetica 18 bold', 15))
horizontal12.place(x=700, y=350, height=200)
```

```
#####
```

```
def createText():
```

```
    t1 = Label(root, text="RGB Led #1", font=("Courier", 20))
    t1.place(x=150, y=0)
```

```
    t2 = Label(root, text="RGB Led #2", font=("Courier", 20))
    t2.place(x=550, y=0)
```

```
    t3 = Label(root, text="RGB Led #3", font=("Courier", 20))
    t3.place(x=150, y=300)
```

```
    t4 = Label(root, text="RGB Led #4", font=("Courier", 20))
    t4.place(x=550, y=300)
```

```
    t5 = Label(root, text="Control Button #1", font=("Courier", 15))
    t5.place(x=70, y=615)
```

```
    t6 = Label(root, text="Control Button #2", font=("Courier", 15))
    t6.place(x=320, y=615)
```

```
    t7 = Label(root, text="Control Button #3", font=("Courier", 15))
    t7.place(x=570, y=615)
```

```
def helloCallBack(btn):
    global buttons

    if buttons != 1 and buttons != 3 and buttons != 5 and buttons != 7 and btn == 1:
        buttons = buttons + 1
    elif (buttons == 1 or buttons == 3 or buttons == 5 or buttons == 7) and btn == 1:
        buttons = buttons - 1

    if buttons != 2 and buttons != 3 and buttons != 6 and buttons != 7 and btn == 2:
        buttons = buttons + 2
    elif (buttons == 2 or buttons == 3 or buttons == 6 or buttons == 7) and btn == 2:
        buttons = buttons - 2

    if buttons != 4 and buttons != 5 and buttons != 6 and buttons != 7 and btn == 3:
        buttons = buttons + 4
    elif (buttons == 4 or buttons == 5 or buttons == 6 or buttons == 7) and btn == 3:
        buttons = buttons - 4

def delay_time():
    time.sleep(1)

def createButtons():

    B1 = customtkinter.CTkButton(master=root, text="ON/OFF", hover_color=("white", "white"),
    text_color=("black", "black"), fg_color=("red", "lightgray"), command=lambda: helloCallBack(1))
    B1.place(x=100, y=650)

    B2 = customtkinter.CTkButton(master=root, text="ON/OFF", hover_color=("white", "white"),
    text_color=("black", "black"), fg_color=("red", "lightgray"), command=lambda: helloCallBack(2))
    B2.place(x=350, y=650)

    B3 = customtkinter.CTkButton(master=root, text="ON/OFF", hover_color=("white", "white"),
    text_color=("black", "black"), fg_color=("red", "lightgray"), command=lambda: helloCallBack(3))
    B3.place(x=600, y=650)
```

```
def turnOnLed():
    global flag

    if flag == b'\xca':
        flag = 0
        updateArray()

    root.after(1, turnOnLed)

def calculate_crc8(data):
    crc = 0x00 # Initial CRC value
    polynomial = 0x07 # CRC8 polynomial (0x07)

    for byte in data:
        crc ^= byte
        for _ in range(8):
            if crc & 0x80:
                crc = (crc << 1) ^ polynomial
            else:
                crc <<= 1

    # Ensure the CRC8 value is only 8 bits (1 byte)
    crc &= 0xFF

    return crc

createButtons()
createText()
createLabels()
turnOnLed()

root.mainloop()
```



## 7.2 Tables

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

**Table 3.1 "Uart\_tx\_Constant" Ports**

Name	Length (Bits)	Description
state_tx	State	Signal representing the current state of the finite state machine
signal_A_q	1	Signal for generate rising edge
signal_A_q_not	1	Signal for generate rising edge
sig_arising_edge	1	Signal indicating the rising edge of "sig_baud_clk"
sig_bit	1	A signal to store one bit of the output "uart_tx_triger"
sig_baud_clk	1	Signal representing the baud clock 38,400Hz
sig_byte	8	Signal representing an 8-bit data byte
sig_packet	12	Signal representing a UART packet
cnt_baud	656	A variable used for counting
sig_cntr	16	A variable used for counting
end_bit	3	Constant representing the end bit = "111"
start_bit	1	Constant representing the start bit = '0'

**Table 3.2 "Uart\_tx\_Constant" Internal Signals**

Name	Type	Length (Bits)	Description
resetsn	IN	1	Asynchronous reset input
sysclk	IN	1	System clock input (operating at 50 MHz)
toggle	IN	1	Input signal used for addressing the RAM
detected_bit	IN	1	Output signal representing the UART transmission trigger
wr_ram	OUT	1	Output signal for controlling the RAM write operation
ram_address	OUT	6	Output signal for addressing the RAM
detected_byte	OUT	8	Output signal for the received byte of data

**Table 3.3 "Uart\_rx" Ports**

Name	Length (Bits)	Description
state_rx	State	A signal of type state, which is an enumeration representing the different states of the receiver
sig_baudx32	1	A clock signal that operates at 32 times the baud rate of 38,400 bps. It toggles every 20 "sysclk" cycles
sig_bit	3	Used for debouncing the received data bit
sig_bouncer_bit	1	A debounced version of the received data bit
signal_A_q	1	Signal used to create a rising edge detector for "sig_baudx32"
signal_A_q_not	1	Signal used to create a rising edge detector for "sig_baudx32"
sig_araising_edge	1	A signal that goes high on the rising edge of "sig_baudx32", indicating the start of a bit
sig_data_bit	1	A signal representing the current received data bit
sig_wr_ram	1	A signal used to control the write operation of the RAM
sig_ram_address	6	A signal used to address the RAM
sig_cnt_address	5	A counter signal used to increment the RAM address
sig_detected_byte	8	A signal used to store the received byte of data

**Table 3.4 "Uart\_rx" Internal Signals**

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

**Table 3.5 "Ram2\_X" Ports**

Name	Type	Length (Bits)	Description
resetrn	IN	1	Asynchronous reset input
sysclk	IN	1	System clock input (operating at 50 MHz)
q_data_ram	IN	8	The data input from ram
biPhase_tx_out	OUT	1	Bi-phase encoded output
start_strobe_tx	OUT	1	Signal indicating the start of transmission
read_address	OUT	6	Provides the address used for reading
rd	OUT	1	Signal indicating a read operation
Toggle	OUT	1	Toggle the MSB bit of address for read/ write mode

**Table 3.6 "BiPhase\_tx" Ports**

Name	Length (Bits)	Description
state_bi	State	A signal of type state, which is an enumeration representing the different states of the "BiPhase_tx"
state_mini	State	A signal of type state, which is an enumeration representing the different states of the "side_state_mashine "
sig_main	14	This signal used to hold a count related to the main clock operation
sig_q_ram_out	8	This signal representing output data from a RAM
sig_read_address	6	This signal used to represent an address for reading from RAM
sig_read_address_cnt	5	This is signal used for counting purposes, possibly related to address generation
sig_shift_data	8	This signal used for shifting data
sig_main_rising_edge	1	This signal indicates the rising edge of the main clock
sig_rd_rising_edge	1	This signal indicates the rising edge of the read clock
sig_main_falling_edge	1	This signal indicates the falling edge of the main clock
sig_read	1	This signal is used to control reading operations
sig_biPhase_tx_out	1	This signal represents the output of the Bi-Phase transmitter
sig_q_data_bit	1	This signal represents a single bit of data from the shifted data
sig_cut	1	Signal used to create a rising and falling edge detector for "sig_main_clk"
sig_cut_not	1	Signal used to create a rising and falling edge detector for "sig_main_clk"
sig_cut_rd_not	1	Signal used to create a rising edge detector for "sig_read"
sig_cut_rd	1	Signal used to create a rising edge detector for "sig_read"
sig_main_clk	1	This signal used to represent the clock derived from "sig_main" counter
sig_inc	1	This signal used to create pulse to start the "state_mini"
sig_toggle	1	This signal is used to separate the sending of data in RAM between reading and writing

**Table 3.7 "BiPhase\_tx" Internal Signals**

Name	Type	Length (Bits)	Description
Resetn	IN	1	Asynchronous reset input
Sysclk	IN	1	System clock input (operating at 50 MHz)
bi_phase_out	IN	1	The input BiPhase signal from card A
signal_out	OUT	1	The filtered output signal

**Table 3.8 "BS\_Filter" Ports**

Name	Length (Bits)	Description
sig_filter	8	A 8-bit vector used as a shift register to store the history of input signals
sig_check_0" to "sig_check_6	1	Individual signals representing the XOR outputs of consecutive elements of the "sig_filter" vector
sig_total_check	1	A signal used to determine the total check result based on the XOR outputs

**Table 3.9 "BS\_Filter" Internal Signals**

Name	Type	Length (Bits)	Description
Resetn	IN	1	Asynchronous reset input
Sysclk	IN	1	System clock input (operating at 50 MHz)
bi_phase_filtered	IN	1	The input BiPhase signal after filtering from "BS_Filter"
main_clk	OUT	1	The clock output signal
nrzl_data	OUT	1	The data output signal

**Table 3.10 "Simple\_BS" Ports**

Name	Length (Bits)	Description
sig_00_clk	1	This signal used to represent the clock derived from counter
sig_00_cut	1	Signal used to create a rising edge detector for "sig_00_clk"
sig_00_cut_not	1	Signal used to create a rising edge detector for "sig_00_clk"
sig_00_r	1	This signal indicates the rising edge of the "sig_00_clk"
sig_90_clk	1	This signal used to represent the clock derived from counter.
sig_90_cut	1	Signal used to create a rising and falling edge detector for "sig_90_clk"
sig_90_cut_not	1	Signal used to create a rising and falling edge detector for "sig_90_clk"
sig_90_r	1	This signal indicates the rising edge of the " sig_90_clk "
sig_90_f	1	This signal indicates the falling edge of " sig_90_clk "
sig_bi_phase_filterd_cu t	1	Signal used to create a rising and falling edge detector for " sig_bi_phase_filterd "
sig_bi_phase_filterd_cu t_not	1	Signal used to create a rising and falling edge detector for " sig_bi_phase_filterd "
sig_bi_phase_filterd_r	1	This signal indicates the rising edge of the " sig_bi_phase_filterd "
sig_bi_phase_filterd_f	1	This signal indicates the falling edge of " sig_bi_phase_filterd "
sig_enable	1	This signal used to represent the enable derived from counter
sig_ff_A	1	This signal represent the samples from " sig_bi_phase_filterd "
sig_ff_B	1	This signal represent the samples from " sig_bi_phase_filterd "
Cnt_clk	14	This signal used to represent counter numbers of "sysclk" to generate clocks or pulses

**Table 3.11 "Simple\_BS" Internal Signals**

Name	Type	Length (Bits)	Description
Resetn	IN	1	Asynchronous reset input
Sysclk	IN	1	System clock input (operating at 50 MHz)
nrzl_in	IN	1	Input signal for data
main_clk	IN	1	Clock signal with a period of 328 microseconds
crc8bit_out	OUT	1	Output indicating CRC result
correlation	OUT	5	Auto correlation checking vector

**Table 3.12 "CRC8BIT" Ports**

Name	Length (Bits)	Description
state_crc	State	The state machine implementation helps organize and control the flow of the CRC calculation process
crc_reg8bit	8	This signal holds the current state of the CRC value as it progresses
sig_sf_reg	32	Serves as a buffer to hold the incoming data stream
sig_cut_main_clk	1	Designed to provide synchronization or timing control within your system
sig_cut_main_clk_not	1	Designed to provide synchronization or timing control within your system
sig_main_clk_f	1	Signal indicating the falling edge of the main clock signal "main_clk"
sig_main_clk_r	1	Signal indicating the rising edge of the main clock signal "main_clk"
cnt	217	This counter helps in controlling the flow of the CRC calculation algorithm

**Table 3.13 "CRC8BIT" Internal Signals**

Name	Type	Length (Bits)	Description
Resetn	IN	1	Asynchronous reset input
Sysclk	IN	1	System clock input (operating at 50 MHz)
nrzl_data	IN	1	Data input signal
main_clk	IN	1	Represents the main clock signal used for timing synchronization within the module
crc8bit_in	IN	1	Represents the result of a CRC-8 error detection calculation
correlation	OUT	5	Auto correlation checking vector
load_leds	OUT	1	Used to control the loading of LEDs, likely indicating when the LEDs should be updated with new data
green_leds	OUT	8	Serves as the data signal for three green LEDs in the system
rgb_leds	OUT	96	Serves as the data signal for controlling RGB LEDs in the system

**Table 3.14 "Data\_Organizer" Ports**

Name	Length (Bits)	Description
state_Do	State	Signal that governs the behavior and operation of the "Data_Organizer" module
sig_main_clk_cut	1	Used to synchronize internal operations with the rising and falling edges of the "main_clk" signal
sig_main_clk_cut_not	1	Used to synchronize internal operations with the rising and falling edges of the "main_clk" signal
sig_main_clk_r	1	Represent the rising edge of the clock signal "main_clk"
sig_main_clk_f	1	Represent the falling edge of the clock signal "main_clk"
sig_sf_reg	32	Serves as a buffer to hold the incoming data stream
sig_cnt	101	Used as a counter to manage the input data shifting process into the "sig_rgb_leds_out" signal
sig_green_leds_reg	8	Serves as a register to temporarily store the input data for the green LEDs
sig_green_leds_out	8	Represents the output data signal for the green LEDs
sig_rgb_leds_out	96	Serves as the output data signal for controlling RGB LEDs

**Table 3.15 "Data\_Organizer" Internal Signals**



Name	Length (Bits)	Description
state_leds	state	This signal represents the current state of the LED controller's finite state machine (FSM)
sig_cnt	100	This signal is used for counting clock cycles or generating timing pulses
sig_cnt_loop	100	This signal is used to keep track of the number of iterations in a loop
sig_shift_led_rgb	96	This signal holds the RGB LED data that needs to be shifted out to the LEDs
sig_OB_LED_RGB_DIN	1	This signal represents the data input for the RGB LEDs
sig_bit	1	This signal holds the value of a single bit from the "sig_shift_led_rgb" signal

**Table 3.16 "RGB" Ports**

Name	Type	Length (Bits)	Description
resetsn	IN	1	Asynchronous reset input
sysclk	IN	1	System clock input (operating at 50 MHz)
load_leds	IN	1	Control signal to load LED data
green_leds	IN	8	Input signal for displaying data on green LEDs
rgb_leds	IN	96	Output signal for displaying data on RGB LEDs
OB_LED_RGB_DIN	OUT	1	Output signal to drive RGB LEDs' data input
LED_1	OUT	1	Output signals for specific LEDs
LED_2	OUT	1	Output signals for specific LEDs
LED_3	OUT	1	Output signals for specific LEDs

**Table 3.17 "RGB" Ports Internal Signals**